PS04CMCA07 The Linux Operating System

Dr. J. V. Smart

Table of Contents

- Vim Editor in More Detail
- Commands List

Syllabus

COURSE NO: PS04CMCA21

THE LINUX OPERATING SYSTEM

w.e.f. June 2017

(3 Lectures & 1 Seminar/Tutorial per Week Total Marks: 100)

COURSE CONTENT:

1. **Introduction to the UNIX/Linux Environment and CLI**
   - Introduction to UNIX and GNU/Linux: history, features, derivatives
   - Overview of different software models and software licenses
   - An overview of the Linux environment
   - Introduction to the Bourne Again SHell (bash)
   - The vim editor
   - Shell environment, commands, syntax, options, getting help
   - File system navigation and manipulation
   - Process management
2. **Basic Shell Scripting**
   - Command line processing
   - I/O redirection and filters
   - The built-in constructs of the shell
   - Basics of filters and regular expressions
   - Basic commands and utilities
   - Examples
3. **Advanced Shell Scripting and System Calls**
   - Using advanced features of the shell
   - The sed filter
   - The awk filter
   - Common Linux commands
   - Programming using system calls under Linux
   - Examples
4. **Linux System Administration**
   - System structure

- Partitioning, formatting, mounting and unmounting file systems
- Devices and file system management
- User management
- System configuration files
- System startup and shutdown, runlevels
- Updating the system and installing and updating packages
- An introduction to the Linux file system

MAIN REFERENCE BOOKS

1. Das S. : Your UNIX – The Ultimate Guide, Tata McGraw-Hill, 2001
2. Nemeth E., Hein T., Snyder G.: Linux Administration Handbook, 2nd edition, Pearson Education / PH PTR, 2007
3. Online Manuals

BOOKS FOR ADDITIONAL READING

1. Kernighan B. W. and Pike R. : The Unix Programming Environment, Prentice-Hall of India, 1994
2. Sobel M.: A Practical Guide to Linux Commands, Editors, and Shell Programming, Pearson Education, 2006
3. Prata S. : Advanced Unix – A Programmer's Guide, BPB Publications, 1986
4. Bach, Maurice J: The Design of the UNIX Operating System, Prentice Hall of India, 1986

The History of Unix

> The success of the UNIX system stems from its tasteful selection of a few key ideas and their elegant implementation. The model of the Unix system has led a generation of software designers to new ways of thinking about programming. The genius of the Unix system is its framework, which enables programmers to stand on the work of others. (the Turing Award selection committee, 1983)

Origins

The Unix operating system is a watershed operating system developed at the then AT&T Bell Laboratories by a group of employees including Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy and Joe Ossanna starting in 1969. In the past three decades, the operating system has had tremendous influence on the way we think about and develop operating systems and computer programs in general. Unix as well as

various Unix-like and Unix-derived operating systems continue to dominate the computer world even today. In the fast changing world of Information Technology, few computer programs have survived for such a long time while still retaining their essential characteristics.

The people who developed Unix were originally working on a joint project of At&T Bell Laboratories, General Electric (GE) and Massachusetts Institute of Technology (MIT) to develop a large ambitious multi-user operating system called Multics (Multiplexed Information and Computing Service). Though the project had many innovations to its credit, it was a large and unwieldy project that was not progressing as expected. When AT&T decided to pull out of the project, Thomson, Ritchie and others decided to develop a smaller operating system to keep alive some of the ideas of Multics. Ken Thomson had developed a game called Space Travel while still on the Multics projects, but it was too expensive to run the game on a large machine in active use. He found a little used machine at Bell Labs and redeveloped Space Travel to run on it. He and his group gradually added the operating system ideas they had in mind and finally came out with a simple operating system that they initially called Unics, because it supported a single user; as opposed to Multics. When it was developed further and started supporting multiple users, the spelling of the name was changed to Unix.

Ken Thompson and Denis Ritchie

Key Success Factors

Unix was the first successful system to have implemented several innovative and revolutionary ideas. Till then, a general consensus was the operating systems can only be developed in an assembly language. Though Unix, too, was originally written in assembly language, in 1972 much of it was rewritten in the C language, a high level language developed specifically for it. Though the idea was not entirely new, it was Unix that popularized this novel concept. Assembly languages are inherently tied to particular machine architecture and it is extremely difficult to port (convert) programs written in the assembly language of one computer into the assembly language of another computer having a very different architecture. On the other hand, high level languages are not tied to any specific hardware architecture and all programs written in a high level language become immediately available on a new computer platform as soon as a compiler for the

language is developed for the new platform. In those days when there were as many different architectures and operating systems as there were computer manufacturers, this easy portability was a major boon. Earlier, people and organizations had to learn a new operating system every time they purchased a new type of computer. Now they had the option of porting their existing operating system to the new computer architecture and continue working in a familiar environment.

Also, as a part of a court settlement with the US Government, AT&T was not allowed to sell computer software. So they did not object to the Unix developers giving out copies of the Unix operating system with source code and online manuals to others for free. Soon many universities, government agencies and private companies started using Unix. Because, the source code was available, it was easy to make the small amount of changes needed to run Unix on a new platform. It also allowed the universities and organizations to study the source code and enhance it with new features. Unix became phenomenally successful in the subsequent years. Denis Ritchie and Ken Thompson were awarded the Turing Award, considered to be the Nobel Prize of computing, in 1983.

The success of Unix can be largely attributed to the revolutionary concepts it pioneered or popularized for the first time. Some of these are listed below.

- Unix was the first successful operating system to have been developed in a high level language
- Unix popularized the multi-level hierarchical file system. With some modifications, it is still in use by all major operating systems
- Unix simplified device access by abstracting I/O to devices also as file I/O
- Unix provided a very powerful command line environment that supported combining the power of existing commands in flexible ways to get a new job done (I/O redirection). This major innovation dramatically changed the way people worked, improved their efficiency and continues to be a major strength of the platform
- Unix stored all configuration information in plain text files, making them easily accessible and modifiable
- Unix started a new trend by providing an online manual with the system itself, so there was no need to walk to the library to fetch a printed manual if one forgot some command or option.
- The communication and networking technologies that are so ubiquitous today, were largely developed first on Unix systems
- Unix provides powerful pattern matching capabilities in many of the tools that are part of the environment

Advantages of Unix

- Multi-user
- Multitasking
- The Unix Philosophy (do one thing well)
- Hierarchical file system
- Emphasis on the use of plain text files for configuration and data storage
- Devices as files
- Pattern matching
- Shell programming language
- Portability
- Online documentation
- Communication
- Security
- Reliability
- Performance
- Scalability
- Abstracted I/O & I/O redirection

from the beginning, Unix has been a system designed by programmers, for the programmers. Unlike other *user-friendly* operating systems that try to protect the user from committing mistakes unintentionally, Unix generally assumes you know what you are doing. It carries out the command issued by the user without bothering him/her with prompts to confirm the action. Unix commands work silently; usually there is no extraneous output when a commands does its job successfully. Thus applies especially to the command line environment. On the other hand, the GUI interfaces are quite user-friendly.

Unix Turns Commercial

As Unix became extremely popular and AT&T, after its breakup, was allowed to sell computer software, the company trademarked the name UNIX and started charging for the Unix operating system. Many computer companies also purchased Unix from AT&T with source code, modified it and developed their own commercial Unix-like operating systems. This generated a backlash among the community that had received Unix for free so far. Some, especially universities, asserted their right to a free Unix system. The University of California at Berkley developed their own free Unix version called BSD (Berkley Software Distribution) Unix. BSD Unix made several important contributions to Unix. AT&T challenged them in court. After a long legal battle, both commercial and free versions of Unix were permitted.

MINIX

The famous academician and author Andrew S Tanenbaum had developed a Unix-like operating system called MINIX for teaching principles of operating systems. Its source code was published as a part of his textbook Operating Systems: Design and Implementation. However, the publisher had restricted its use to those who purchase the book and to academic use only. Thus, it was not free.

Andrew S Tanenbaum

The Genesis of the Free Software Movement

Around early 1980s, Richard Stallman, working at MIT, got increasingly frustrated by the various restrictions placed by commercial software vendors on use and sharing of computer software. His vision for software was to provide all kinds of freedom to the users. His interpretation of the word free was not free as in free buttermilk i.e. gratis (without payment of money), but it was free as in freedom. He outlined four fundamental types of freedoms for users of software.

- **Freedom 0 :** The freedom to run the program, for any purpose
- **Freedom 1 :** The freedom to study how the program works, and change it so it does your computing as you wish. Access to the source code is a precondition for this.
- **Freedom 2 :** The freedom to redistribute copies so you can help your neighbor
- **Freedom 3 :** The freedom to distribute copies of your modified versions to others

Richard Stallman at an Inauguration in Kolkata

To this end, he started the GNU project in 1983. He wanted GNU to be a Unix-like (in its working), but completely free operating system. To emphasize that GNU was not a commercial system like Unix, he chose the name GNU that stood for GNU is Not Unix. This is called a recursive acronym. Such recursive acronyms are common in the names of computer programs, especially open source ones. By free, he meant all the freedoms mentioned above. He started a Free Software Foundation (FSF) for developing GNU and other entirely free software projects. By 1990, he and the volunteers of the FSF had created most of the major components of the proposed GNU operating system, including the compiler, the shell, the libraries and the utilities. But the core component, the kernel, was unfinished. Work on the kernel was going on, but was slow.

Linux is Born

> I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu)… – Linus Torvalds (1991)

In 1991, Linus Torvalds, a student from Finland, developed an experimental operating system kernel for the PC (personal computer) called the Linux kernel (Linux stood for Linux is Not Unix, again a reference to Unix's now commercial status). After he opened communications with other programmers on the Internet, the project grew rapidly. With the help of these volunteers, finally the Linux kernel and the GNU components were combined to form the first completely free working Unix-like system (BSD's case was still going on in the courts). The combined system came to be known as GNU/Linux, or simply Linux and became widely popular after Linus granted all the freedoms to everyone (GNU had already done so) and a large community of volunteers, individuals, organizations and even commercial corporations, started supporting its development.

Linus Torvalds

It’s All About Freedom and Choice

Linux is a mass movement today. A large community spearheaded by Linus Torvalds himself looks after the development of the Linux kernel. Several other individuals and communities continually work towards providing better software solutions for all common requirements of users with liberal freedoms. In most cases, the freedom includes the rights to obtain the source code of the software, modifying it and contributing the changes back to the community or redistributing the modified software under a name of one’s own choosing. As a result, there is a bewildering choice of software available for all the common requirements of computer users. Different people and organizations select bundles of software from this vast pool of software as per their own criteria and preferences and create a distribution of the Linux operating system that they distribute under their own name. Each distribution of Linux is a bundle of some version of the Linux kernel and a set of software applications selected with some goals in mind. There are thousands of Linux distributions, called distros, available now. Some of the more popular ones include Ubuntu Linux, Linux Mint, Fedora Linux, Debian GNU/Linux, Red Hat Enterprise Linux (this is a commercial, but open source, distribution), openSUSE Linux, Knoppix, etc. These distribution vary in their goals as well as contents. The range include from the 12 MB Tiny Core Linux and 100 MB Puppy Linux to the large ones reaching gigabytes in size and bundling practically everything you may need, and then some more. All of them are variants of the same operating system, because all of them run the Linux kernel. However, due to differences in the selection of the window manager and shell, the looks may differ markedly. Some other operating systems also have some of their major components based on Linux or Unix. For example, both the OS X operating system for

personal computers and the iOS operating system for smartphones and tablet computers from Apple Inc. use a BSD Unix subsystem. The Android smartphone operating system from Google uses a slightly modified Linux kernel as well as several Linux libraries.

Free v/s Commercial Software

By now the reader might be wondering why even commercial organizations get involved in the development of software products that are avowedly free. They not only encourage their development from the ringside, they even commit their resources (programmer time as well as money) to such products. Some companies like Red Hat Inc. and Canonical Ltd. exist primarily for developing such products. To understand this, one first needs to study the cost and revenue dynamics of the software world. Let us have a look at both the commercial software model and free software model.

Commercial Software Model

Developing computer software costs; and it costs a lot. In most projects of automation the software cost is substantially higher than the hardware cost. Much of this cost is due to the salaries paid to skilled Information Technology (IT) professionals. Software development is a complex process and risk of failure is also quite high. Under the circumstances, commercial organizations developing software expect substantial returns on their investment. However, once a single copy of the software goes outside the organization, it is ridiculously easy to make any number of copies of the software. This is where the software business is completely different from the brick-and-mortar businesses. In product manufacture, even if you get access to the blueprint for making a product, it is not easy to produce a genuine looking duplicate product because one would also need matching material and machines. On the other hand, while the process of creating the first copy of software is very costly, the process of creating additional copies from it cost next to nothing. Hence software companies try to prevent their customers from making copies of the software and start selling the software themselves.

Usually, the software companies do not provide the original source code developed by their programmers in a high-level language at great efforts to their customers. They only provide the binary machine code or executable version of the programs to the customers. While this is good enough to run on a computer, it is extremely difficult to understand and modify such binary versions of software. If someone takes a company's software and sells it without any changes as one's own development, the bluff will get caught. If the original developer has asserted one's copyright on one's original work of creation, the copyright laws prevent others from making copies of the work without permission from the copyright holder.

Further, the companies never “sell” the software, they merely license it. Even when you “purchase” legal software, actually you do not purchase the software; you merely purchase a license to use the software subject to a host of terms and conditions. That is why, when you install some software, you are made to click on an “I agree” or “Accept” button. Once you click on this button, you are legally bound to follow the licensing terms that, among other things, prohibit you from making copies and giving to others. A host of other measures like registration keys and Internet based activation are used to prevent piracy (illegal copy and use) of software. Despite all these efforts, piracy does happen on large scale, especially in certain parts of the world and companies have to recover the lost revenue from the genuine purchasers of software.

Free Software Model

There is another side of the coin also. There is a large number of highly skilled persons having an altruistic nature who are ready to spend some part of their time creating free software or helping others out in developing software, without expecting anything in return. Large communities of such “Good Samaritans” collaborating over the Internet can develop (and indeed have developed) even very complex software that would have been quite costly otherwise. Since these people also provide the source code with the software, others get a chance to easily modify the same, correct errors and add new features. Over a period of time such software may not just rival, but even surpass the equivalent commercial offerings. In this environment, there are no restrictions on anybody and everyone enjoys full freedom regarding what one wants to do with the software. This is the vision of Richard Stallman and other advocates of free software. According to Stallman free software does not mean just that you don’t have to pay for it. Free, in his version, stands for freedom – you are free to do whatever you want to do with the software; you may use it, you may give it to others legally, you may study its source code, you may modify it and redistribute it under your own name; you have almost all the freedom. However, as a moral duty you should give credit to the original developers and hence you must preserve their copyright notices, while adding your own (even free software contains copyright notices, if only to ascertain the authors and to prevent others from claiming a right on similar work). Also, if you modify the software, you may persuade the original developer to incorporate your modifications in the original software or you must distribute the modified software under a different name; because if any errors were introduced by your modifications, you should be responsible for the same, not the original developer.

Richard Stallman has also advocated another condition on such free software – if you develop some modified or new software based on such free community-developed software, you must also give the software you developed in this way (with source code)

back to the community for free. This, according to him, is the only way to sustain the free software movement and continue further development of community software. Software given for free under this condition is called “copyleft” (the exact opposite of copyright). He has designed a software license called GNU GPL (General Public License) that enforces all these conditions and he exhorts everyone to release their software under this license. The license means that if you take some code under the GPL and modify/enhance it, you must distribute your enhanced/modified code also under the same license i.e. the GPL. In other words, you must also provide your modified software along with its source code freely. Hence a commercial entity cannot take *GPLed* software, enhance it and sell the enhanced version commercially to make money. There are groups that do not like such strict “copyleft” restriction. They distribute their software under more “permissive” licenses like the BSD or Apache license that do allow commercial entities to sell software made or enhanced from free software. In a multi-licensing scheme, the software is distributed under more than one license and the user is free to select which license they want to obey.

Different Types of Free Software

Free software come in many different flavors – GPL provides the highest freedom to the users, and also, indirectly, prohibits commercial use. The GPL is considered to be the purest form of free software. Software whose source code is freely available is called open source software. *GPLed* software must also provide the original source code of the software and hence is also open source. Sometimes the binary machine-executable version of the software is provided for free, but the source code is not provided; so modifications cannot be made. Such software is free, but not open source. Sometimes the software is given for a free trial for a certain period or with certain limitations or certain functionalities disabled. If you like the software in the trial, you are supposed to purchase it. The software may stop working after the trial period. This is known as shareware or trialware. Sometimes, even if you do not pay, the software may continue to work fully, but with annoying reminders about the pending payment with a waiting period of few seconds or a minute or two. Such software is sometimes known as nagware.

Sometimes the full software is made available for free, but under some legal restriction such as “for personal/academic/non-commercial use only” and for commercial use one must purchase the software. Some software is free and development is supported by voluntary donations. Some software developers earn from the revenue of advertisements shown in the software or on the maker’s web site. Such software is known as adware. Some software is made available for free, but you are requested to pay or donate a small amount for its use and to support further development. However, if you do not pay, the software continues to work fully.

The Free Software Revenue Model

While companies selling proprietary software earn from the sales revenue, companies working with free software of various categories have different revenue models. Some of them earn money by providing “training and support services” on a commercial basis. Due to the complexity, large software often has several bugs or errors in them. An individual encountering such errors may not suffer much materially. The person may get help from friends, from Internet forums and try to solve or bypass the error on one’s own. This may sometimes take a long time also. But for commercial organizations if the computer systems don’t work for some time, it may incur heavy financial losses. Hence they prefer the assurance of getting training and support from qualified engineers, often the very people who have developed or contributed significantly to the software. Such organizations are even ready to pay for such support, if it turns out to be cheaper than buying proprietary software. Hence giving the software for free to everyone and charging corporate entities for support is one strategy. Canonical Ltd. has followed this strategy for Ubuntu.

Other strategies include displaying online advertisements and earning revenue from them, giving a basic version free and charging for a “premium” version with more features, etc. The latter model is named "freemium". Companies can even make money by collecting contact details (like email id) through a “free registration” process and then selling these contact details to mass advertisers. Some companies support a completely free and open source version of particular software, but derive indirect benefits by incorporating the enhancements made to the free version by the community into their own commercial offerings. For example, Red Hat Inc., another major Linux company, started and supports the free and open source Fedora Linux distribution in a major way. Enhancements made in Fedora Linux by the collaborative work of Red Hat engineers and thousands of volunteers around the world often find their way into the commercial Red Hat Enterprise Linux product.

> Do you still think the commercial interests of a for-profit company and free products are just not compatible with each other? Consider these facts –
>
> - On 19th January in the year 2000, shares in Yahoo! Japan became the first stock in Japanese history to trade at over ¥100,000,000, reaching a price of 101.4 million yen ($9,62,140 at that time) for a single share. All of the company’s products and services were free.
> - In the year 2016, Google’s revenues from advertising stood at $79 billion.
> - On 18th May 2012, when Facebook entered the stock market for the first time, the company was valued at $104 billion. The company provides free social networking services to its users.

Proprietary software v/s open source software

- Cost
- Freedom
- Bugs, patches, vulnerabilities
- Features
- Compatibility
- Choice

Why commercial organizations take part in free software development

- Earning by providing support
  - Total Cost of Ownership (TCO) Cost of purchase, maintenance, training
- Advertisements
- Profiling users and collecting personal information
  - Targeted advertisements
  - Machine learning
- Parallel versions (e.g. RedHat Enterprise Linux and Fedora Linux)
  - Community edition v/s enterprise edition
- Benefiting from side effects (e.g. Android)
- Developing software for internal use and then releasing as open source

Different categories of free software

- Free and open source software (FOSS)
- Free but not open source
- Trial versions (limitation - time, features)
- Full, but legally limited versions
- only for non-commercial use
- Shareware
- Adware
- Nagware

Different types of licenses and IPR (Intellectual Property Rights) issues

- IPR
  - Patent

    - Copyright
    - Trademark (logos, names, distinctive style)
- Proprietary licenses
- Free licenses
    - Copyleft / restrictive licenses (GPL)
    - Permissive / liberal licenses (Apache, BSD, MIT, Mozilla, etc.)
    - Creative Commons licenses
- Other intellectual assets
    - text
    - images
    - audio
    - video
    - animation

Comparison of Popular Operating Systems for Personal Computers

When we compare popular operating systems for the personal computers, several differences prop up. There are three major families of operating systems for the PC. The proprietary Microsoft Windows family of operating systems is by far the most popular. Apple Computer's Mac series computers use their own proprietary operating system called OS X (earlier it was the MAC OS). These computers have found a niche in the premium brand-conscious segment where they have a fierce fan following. The third operating system is Linux. Linux is used by a variety of people, from the geeks (very technically minded people) to people who want to support the free software movement to people owning older or lower configuration hardware that won't run the other operating systems properly or people who can't or don't want to pay for an operating system. Each OS has its own pros and cons.

Microsoft Windows

Microsoft Windows is a proprietary OS and must be purchased for using it. It is a very user-friendly, easy to use OS and almost everyone is familiar with it. Because of its popularity, a large number of commercial as well as free software is available for the OS. It also boasts of an excellent device driver support. Even though Microsoft has taken significant strides in improving its stability and security compared to early versions, lingering doubts about these issues still remain. Being the most popular OS among users has also made it the most popular target of attacks by crackers (highly skilled programmers with malicious intention). Malware like computer viruses, worms, Trojans

and several others continue to affect systems running Microsoft Windows. Even using a top-notch commercial security solution does not seem to be able make one's system completely immune to such attacks. Security of Microsoft Windows systems is a continuous headache for large organizations. Microsoft Windows is also infamous for needing too much hardware resources and higher-end configurations to run decently. As people run a mix of software from so many different sources on Microsoft Windows systems, the overall experience is somewhat varied and when there is a problem, it becomes difficult to pinpoint the exact source of the trouble. Microsoft Windows officially supports only the built-in shells for the GUI (Windows Explorer) and CLI, though third party alternative are available.

Apple OS X

Apple OS X is also a proprietary operating system. It comes bundled with the machine and neither work without the other (barring some third party experimental solutions). The system is known for its high quality hardware and visual appeal and what ardent Apple fans believe to be the best user experience. The entire hardware, operating system and software environment is tightly controlled by Apple to provide a highly consistent and reliable user experience. OS X also officially supports only the built-in GUI and CLI, though third party alternatives are available to try. Though the systems are considered quite secure; it, too, is not immune from attacks. The major advantage of the system is its consistent, high-quality user experience. Against that, the user is confined to a narrow world controlled by Apple and third party application support is limited. Also, the product comes at a high premium. Internally, OS X is a graphical environment that runs on top of a Mach kernel and a BSD Unix subsystem.

Linux

Linux, just like its predecessor Unix, is known for its high performance, security, reliability and portability. No matter how antiquated or low-end the hardware is, one can find a Linux distribution that would run on it. Even though lack of whole-hearted support by BIOS and device vendors in providing high quality device drivers dents Linux's image of being a reliable and portable OS to some extent, it is quite stable on most PC configurations. It combines the high-power CLI that has traditionally been Unix's strength with an impressive GUI that makes it almost as user-friendly as the other two OSs. Linux lets us choose from a wide variety of distros, then presents an even wider array of mostly free applications to install from and is flexible enough that with experience we can configure it exactly the way we want it to be. However, free applications available under Linux sometimes lack the polish of corresponding Windows applications (especially on the GUI front) and often may not provide all the bells and whistles that commercial

applications on Microsoft Windows or OS-X platforms provide. But they do get the job done most of the time and keep getting improvements from the community. Upgrades are as free as the base OS and it is just a matter of user's convenience when they want to upgrade their systems. However, one should be prepared for the occasional glitches. Linux provide a number of choices for the CLI as well as GUI. The most common GUIs are KDE, GNOME and Unity. The newest versions of the GUIs provide a strong competition to the other two OSs as far as visual attractiveness is concerned, though they, too, require a bit higher configuration.

A major advantage of Linux is that not only is it free; it also provides us complete freedom in running the operating system. Unlike other operating systems that require at least one free hard disk partition (that would be erased during the installation) and a tedious installation process, there is a range of choices for running Linux – from a no-risk no-installation trial using a Live CD to installing Linux alongside Windows (multi-boot configuration) to booting Linux directly off a USB flash disk or even the network – Linux supports it all. Linux also provides excellent interoperability with Microsoft Windows and you can access the Windows partitions on your computer and the organization's Microsoft Windows servers as easily as from Windows. Unfortunately, facility of accessing the Linux partition from Microsoft Windows is generally limited to read-only access (we can read the files but cannot modify them) using third party tools. The office suite on Linux – OpenOffice.org or LibreOffice provides good interoperability with the Microsoft Office suite with few caveats and some irritants. In essence, Linux meets all the basic needs of an average computer user and, once you settle down, can be as comfortable to use as it can get. And all this is yours, 100% free, 100% legal, no strings attached.

The market share of Linux is difficult to estimate because it is free and there is no central point of distribution, but Linux has a very dominating presence in supercomputers, mainframes, servers and graphics workstation markets. While much of the PC market is dominated by Microsoft Windows and OS X has a considerable presence in the premium segment and has its own loyal fan base, Linux is strong in some organizational setups and at the lower end of the spectrum, with low-end PCs. It is also popular among developers. Linux, too, has developed its own fan base who sneer at people using any other operating system.

Introduction to Ubuntu Linux

> A traveler through a country would stop at a village and he didn't have to ask for food or for water. Once he stops, the people give him food, entertain him. That is one aspect of Ubuntu, but it will have various aspects. Ubuntu does not mean that people should not enrich themselves. The question therefore is: Are you going to do

> so in order to enable the community around you to be able to improve? – Nelson Mandela (anti-apartheid crusader, freedom fighter and former president of South Africa)

Ubuntu Linux is a Linux distribution created by the UK based company Canonical Ltd., established by the South African entrepreneur Mark Shuttleworth. It is in turn based on the Debian GNU/Linux distribution. Ubuntu is an ancient African word meaning 'humanity to others'. It is a philosophy that emphasizes putting common goals and the community above individual interests and believes in helping one another. As the open source software community also has similar philosophy, Ubuntu was chosen as the name of the distribution (if you want to know one more reason for choosing this name, read the two words after the second “the” in the first line of this paragraph again). Ubuntu is free and open source software. Canonical expects to earn money by providing paid support services. While Canonical is the main sponsor, Ubuntu is also supported by the Ubuntu Foundation and large developer and user communities. Ubuntu focuses on usability, security and stability. Its focus on usability (ease of use) and good device support has allowed it to gain and retain a place among the top Linux distributions.

Ubuntu Versions

Ubuntu has a fixed release cycle, with a new version being released in the April and October months of each year. The release numbers are denoted by two-digit year, followed by a dot, followed by two-digit month. Thus, Ubuntu 12.04 LTS was released in April 2012. The releases also have two-word names, with the first word being an adjective and the second the name of an animal. For example, Ubuntu 12.04 LTS was called Precise Pangolin, Ubuntu 12.10 was called Quantal Quetzal. Ubuntu 16.10 is called Yakkety Yak. People often use only the first word to identify the release. The first words are chosen in alphabetic order, so one can know which version is newer just by looking at the first letter of the name. Each desktop edition release is officially supported for 9 months. Every two years, a Long Term Support (LTS) version is released. These versions are supported for 5 years.

Version	Code name	Release date	Supported until
14.04 LTS	Trusty Tahr	2014-04-17	2019-04
14.10	Utopic Unicorn	2014-10-23	2015-07
15.04	Vivid Vervet	2015-04-23	2016-02
15.10	Wily Werewolf	2015-10-22	2016-07

Version	Code name	Release date	Supported until
16.04 LTS	Xenial Xerus	2016-04-21	2021-04
16.10	Yakkety Yak	2016-10-13	2017-07
17.04	Zesty Zapus	2017-04-13	2018-01
17.10	Artful Aardvark	2017-10-19	2018-07
18.04 LTS	Bionic Beaver	2018-04-26*	2023-04

There are several other variants of Ubuntu including a server edition, Edubuntu (a distro aimed at school students, with built-in educational software), Kubuntu / Ubuntu GNOME (variants using the KDE / GNOME desktop instead of the default Unity desktop), Lubuntu / Xubuntu (lightweight variants for use on low-end computers using the LXDE / Xfce desktop environment), etc.

Strengths and Weaknesses of Ubuntu

The key strengths of Ubuntu are usability, good device support, support of large user and developer communities and adoption by many manufacturers as well as some large organizations. A constant complaint against Ubuntu is that it does not give back to the community as much as it takes from it and often deviates from common interests to chart its own path. Canonical's commercial interests are also sometimes a topic of debate.

The Present Landscape of Unix / Linux Derivatives

Many derivatives of Unix and a large number of distributions of Linux are in use today. The common Unix derivatives include BSD Unix, FreeBSD, OpenSolaris, HP-UX, etc. There are many Linux distros and their popularity is tracked by websites such as distrowatch.com.

Pros and Cons of using Linux

Linux is a free and open source operating system that gives its users all the freedoms. It is known for its performance, security, scalability and portability. it provides an excellent environment for development. It provides a powerful command line environment as well as an easy to use graphical environment. It also has good support for the PC hardware. There is a large community of users who are ready to offer help without expecting

anything in return. it has a vast collection of free and open source software that makes it a perfectly usable system for most use cases. Its high performance, scalability and security make it an ideal choice for servers. Due to its ability to work remotely through command line, availability of powerful open source development platforms and low hardware requirements have made Linux the operating system of choice for cloud deployment also.

On the other hand, naive end users often complain that Linux is not as intuitive or easy to use as its proprietary alternatives. While the community provides good support for major issues, minor issues are often relegated to the back burner. Paradoxically, freedom provided by open source software is also a problem. Software changes too fast, leading to a rather steep learning curve. There is too much choice for some types of software and the preferences continually shift.

Linux is a good platform for those people who have an ideological preference for freedom and openness and who are willing to put in a little effort in learning new things. It is also a very good platform for servers, development systems and cloud deployments.

Pros and cons of using Linux

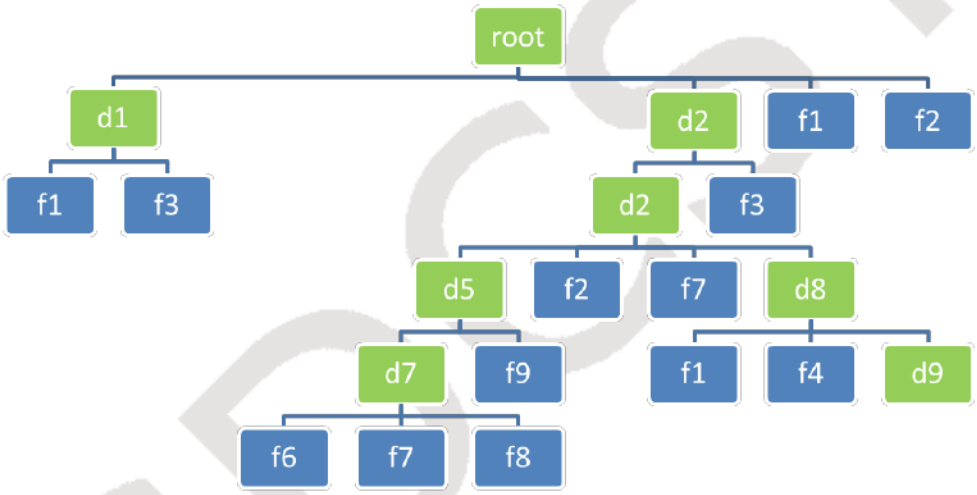
- Cost
- Flexibility
- User-friendliness
- Power
- Compatibility with hardware, device drivers
- Compatibility with Windows software (wine, mono)
- Security
- Collaborating with Windows-based PCs and Servers
- Developed for the programmers by the programmers (CLI)
    - Assumes that you know what you are doing

The File System

The general purpose computers also have storage facilities. The secondary storage is organized in blocks and, like everything else with computers; the blocks are accessed by block numbers. Today's hard disks may have over a billion blocks. This method, though natural for computers, is quite problematic for humans. It is impossible for a human being to remember what was stored in which block number. When some piece of information is large requiring many blocks, one must remember a series of block numbers (that need not be sequential). There is no marking on the blocks to indicate which one is in use and which one is not. One must be careful not to commit the mistake of writing some data to

a block to which earlier some other useful data was written; as in that case the previously written data would be silently overwritten and lost. In case of the same system being used by multiple users, of course, there is no way for one user to know which blocks are in use by the other users.

To avoid all these problems, the operating system provides a file system interface to secondary storage. The concept of a file system is modeled after the filing cabinets commonly found in offices, but with new twists. A file system chiefly contains two types of objects – files and directories (also known as folders). A file is the basic unit of secondary data storage on computers. Any data that the user wants to store will go in some file in the file systems. The files are identified by their names, which are much easier for humans to remember than block numbers. As a disk may have a large number of files, directories are used to organize them. A directory is nothing but a container that may contain files as well other directories known as its subdirectories. The subdirectories may, in turn, contain files as well as subsubdirectories and these may contain files as well as further directories and so on. In fact, there is no theoretical limit to such nesting (putting one container object inside another). However, every file system starts with what is called its root directory and then the root directory may contain files as well as subdirectories and it can go on and on like that. The following figure illustrates this concept. Here the green nodes represent directories, while the blue ones represent files. As an added convenience, directory names begin with d (except root) and filenames begin with f, though there is no such rule.



The file system structure

Internally, the data is still accessed by disk block numbers, but now the file system component of the operating system keeps track of which file or directory is stored in

which block number(s) and which blocks are free and which ones are in use. This takes a great burden off the human users. To maintain this information, some data structures are created on the disk.

The file systems follow certain basic rules. Each file system has a single root directory that is the starting point of the file system. Each directory in the file system contains a number of objects (files and directories), each of which must have a name unique within that directory. Thus, there can never be two objects with the same name in the same directory. However, two different directories can have two different objects with the same name. As long as you know the directory containing the object you want to use, knowing its name may be enough as it is guaranteed to be unique within its directory. Otherwise, you will have to specify the absolute (or full) path leading to the object. That is, you start with the root directory, then the subdirectory, then the sub-subdirectory, and so on, until you reach to the directory that contains the object in question and finally the object itself. Each of these components is separated by a special character that is not permitted in file or directory names. This path uniquely identifies an object in the entire file system.

The file system structure is generally described as a tree structure, with the root directory forming the only root of the tree, all other directories forming branches and the files forming the leaves. If a directory is contained inside another directory, the former is said to be the child of the latter and the latter is said to be the parent of the former. Every directory, except the root directory, has a parent directory. The files cannot hold further objects (files or directories) in them, and hence are the end of the branches. They are known as leaves. With computers, tree structures are typically drawn upside down, with the root at the top.

A blank disk initially contains no file system. The initial blank file system structure (empty data structures) is created by an operation known as formatting the disk. Formatting a disk that already contains a file system destroys the existing file system and replaces it with a new blank one. Certain types of disks such as the hard disks can contain multiple partitions. Each partition may be formatted to contain a separate and independent file system. Formatting one partition does not affect the others. The operating system provides utilities for viewing, modifying and formatting the disks and their partitions.

The command line interface provides commands to navigate the directory structure, while the graphical environment provides a file browser to explore the file system. Common operations provided at the file system level with both the user interfaces include creating new files and directories, copying a file or directory to another part of the file system, moving a file or directory to another part of the file system, renaming a file or directory, deleting (removing) a file or directory (including all the contents), etc. Often security constraints may be in place to prevent a user from carrying out certain operations on

certain files or directories.

Different devices and different operating systems use different file systems. For example, Microsoft Microsoft Windows family of operating systems typically use a file system called FAT (File Allocation Table) along with its variants and a file system called NTFS (New Technology File System). The first one is very simple in nature, but limited in power and performance. It also does not provide any security. The latter is far more advanced than the former and provides a host of features, including security. Both file systems are case insensitive, i.e. capital letters and small letters are treated as same. Thus, you cannot have two files with the names myletter and MyLetter in the same directory. Both generally divide the filename into two parts – the filename proper followed by a . (dot) and the extension. Both use the \ (backward slash or backslash) character as the path separator character. The extension part of the filename is used to signify the type of the file. Different Unix systems use different file systems. Linux generally uses some version of the extended file system (ext2, ext3 or ext4), though several others are also available and in use. These are quite powerful and feature rich file systems. The extended file system is case sensitive, i.e. capital and small letters are treated as two different characters. So you may have two files with the names f1 and F1 in the same directory. It uses the / (slash) character as the path separator. The concept of using the extension part of the filename to signify its type is not mandatory and is weakly used. USB flash disks and memory cards used with mobile phones usually come formatted with the FAT file system when the size is upto 32GB and the exFAT file system when the size is > 32GB (for example, the micro SDXC cards). CDs generally use the ISO9660 file system, while DVDs use the UDF file system. These different file systems have different characteristics.

The Unix/Linux File System

- Hierarchical file system
  - While the Windows file systems is actually a forest, the UNIX file systems is a proper tree (actually a directed acyclic graph)
- The root directory is identified by the / (slash) character
- Directories and files
- Identifying a file system object uniquely using path
- Components of the path are separated by the / (forward slash) character
- / (forward slash) v/s \ (backslash)
- Filenames are case sensitive
- Extensions not an essential part of file name (executable files usually have no extension)
- Any character except / can be used in file names
- The notion of current directory and relative paths

- Absolute path v/s relative path
- Interactive users have a home directory. A user has full permissions on one's home directory. The ~ (tilde sign) is a shortcut for the user's home directory in file system manipulation commands
- Hidden files and directories - any file or directory whose name starts with a . is considered hidden. Hidden files and directories are not displayed in the nautilus file browser (GUI) or by the ls command (CUI). However, there are options to display them

Accessing Multiple File Systems

A computer system may have many storage devices in it. Also, removable devices may be inserted or attached and removed at any time. Each device has its own file system on it. A device that can have multiple partitions, like the hard disk, has a separate file system on each partition. How does one access these file systems? Operating systems like Microsoft Windows assign a separate drive letter (like C:, D:, E:, etc.) to each file system. However, Linux and other Unix-like systems have a single file system tree starting with the root directory, denoted by / (the slash character). The file system contained on the partition from which Ubuntu boots is called the root file system. The root directory of this file system becomes / - the root of the entire file system tree. Initially this is the only file system available.

We may access any other file system by mounting it on any existing directory (this directory is called the mount point). Once mounted, the contents of that file system appear as the contents of the mount point directory. If the mount point previously contained some contents (files and subdirectories), they are masked (hidden) for the duration of the mount. Now we may access (and modify) the contents of that file system from the mount point directory. When we no longer need to use the file system, we may unmount it. At this point, the original contents of the mount point directory get unmasked (become visible again). This process is depicted in the following figures a, b, c and d. Figure a shows the root file system. Figure b shows the file system on another device. Figure c shows the situation after mounting the file system of figure b onto the directory d3 of Figure a. The original contents of d3 are now masked and the contents of the file system mounted there appear as if they are the contents of d3. Figure d shows the situation after unmounting the second file system. The original contents of the directory d3 now become visible again.

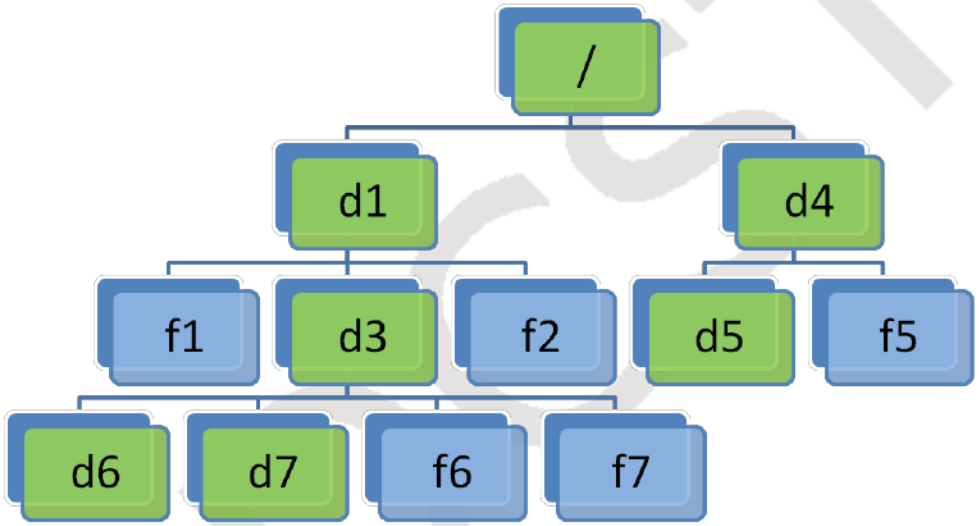


Figure a: The root file system

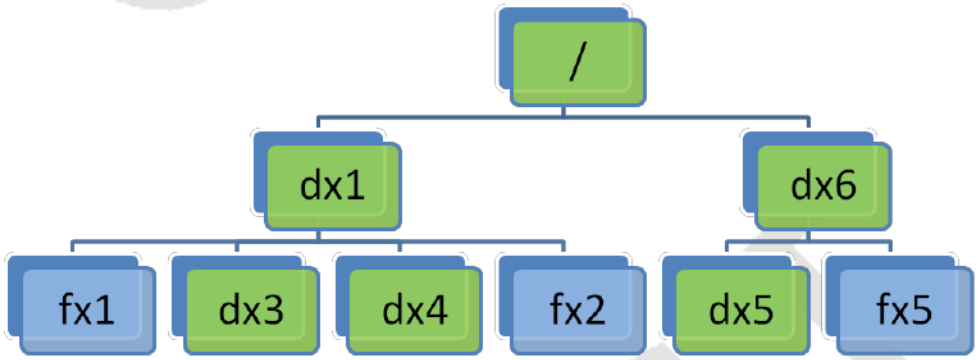


Figure b: File system on another device

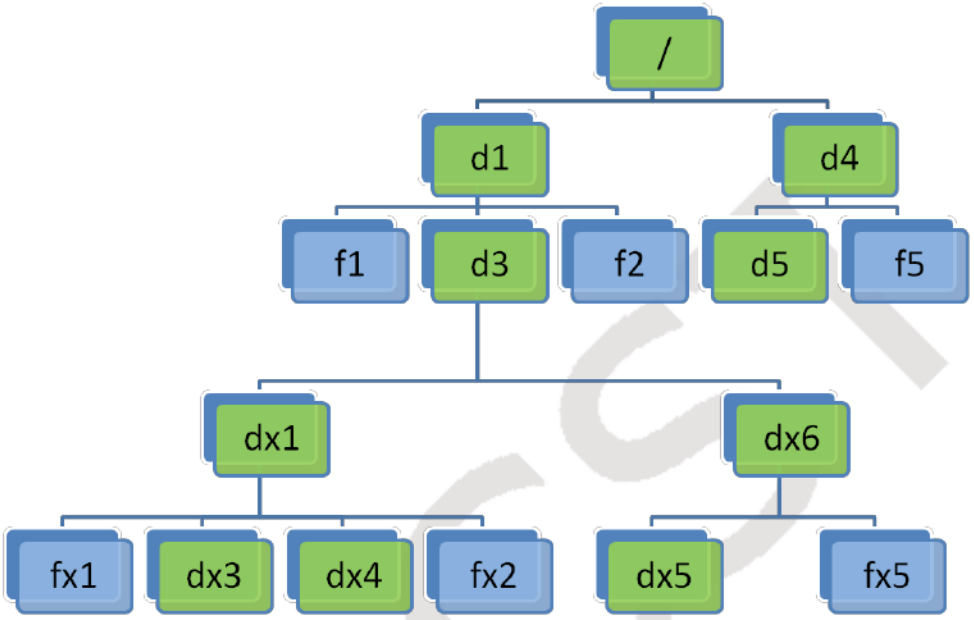


Figure c: After mounting the second file system on directory d3

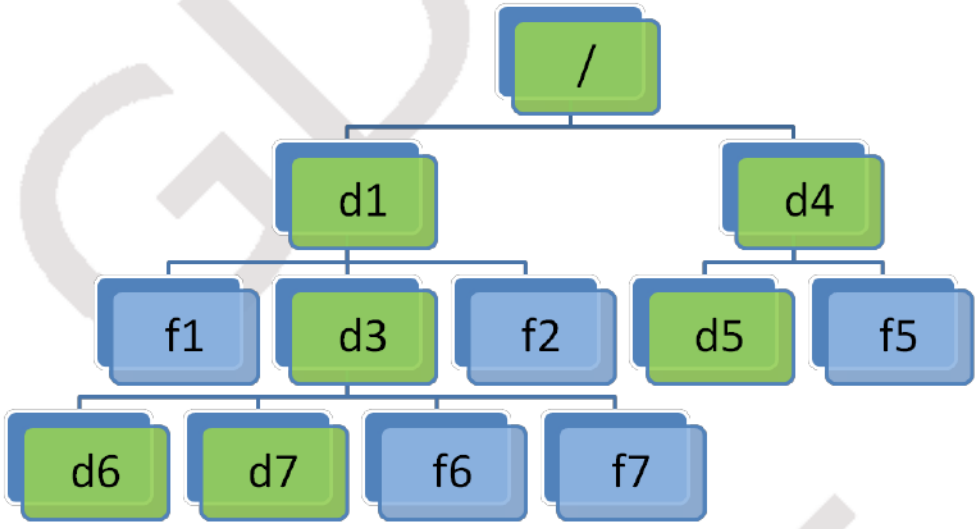


Figure d: After unmounting the second file system from directory d3

However, the common practice is to mount file systems onto empty directories. In the default configuration, Ubuntu automatically detects other fixed devices in the system and shows them in the places menu and the left pane of the file browser. They are mounted when we first try to access them. The file browser shows a triangular icon alongside all mounted file systems other than the root file system. The file system can be unmounted by clicking on this icon. It may be mounted again when the user tries to access it again next time. The root file system cannot be unmounted. Removable devices are automatically mounted when they are inserted. Read-only media like optical disks can be

unmounted by simply removing or ejecting them. Media on which writing is possible (like USB flash disks) must be unmounted by clicking the unmount icon in the file browser or by right clicking its icon on the desktop and selecting “safely remove device” option. This causes any data cached in main memory for improved performance to be flushed to the disk. A message at the end of this process announces that it is now safe to remove the device and the unmount icon disappears from besides the device’s entry in the file browser’s left pane. Only after this the device can be safely detached from the system. Failure to observe this procedure may result in loss of data or damage to the file system on the removable media. While this method of accessing the other storage devices in the system may sound unnecessarily complicated, it is more flexible and powerful and has several distinct advantages.

Not only local file systems, even the remote file systems can also be accessed in the same way. For example, Windows shares can be mounted on local directories using the SMB (Server Message Block) protocol and then be accessed just like local content. Similarly, NFS (Network File System), FTP server directories can also be mounted on a directory and accessed as a part of the file system.

Of course, all these procedures can also be carried out using commands. It is also possible to configure the system to mount some file systems at particular mount points automatically every time the system boots. By default, the system mounts other fixed and removable devices in the system in directories under the /media directory.

User Accounts and Home Directories

There are different kinds of users in the Linux system. *Normal* users can control their own files and settings, but cannot make system level changes. *Administrative* users can access files belonging to other users also. They can also make system level changes. An *interactive* user is a user who can login to the system through the (text mode or graphical) terminal interactively. A *non-interactive* user account is created and used by the operating system or some other software package internally. Such users are not allowed to login to the system interactively. Every interactive user has a home directory. A user has full permissions on one's home directory while other normal users have no permissions on it (a user with administrative permission can still access it). In paths, the ~ (tilde) character represents the currently logged-in user's home directory

User accounts are identified by their names by the human users of the system; but internally, they are identified by an integer user id. There is a special user account with numerical user id 0, usually called *root*. This user is the most powerful user in the system.

The Standard File System Layout

- **/bin** (binary) This directory contains the standard programs that may be useful to all the users of the system
- **/boot** This directory contains the files used for booting the system
- **/dev** (device) This directory contains the files corresponding to various hardware devices
- **/etc** (et cetera) This directory contains all the system-wide configuration files
- **/home** This directory contains the home directories of interactive users (except *root*)
- **/lib** This directory contains the system libraries
- **/lib32** This directory contains the 32-bit system libraries on a 64-bit system
- **/lib64** This directory contains the 64-bit system libraries on a 64-bit system
- **/media** This directory is used to mount removable drives automatically when they are inserted
- **/mnt** This directory is used for additional file systems that are to be mounted on a permanent basis (at boot time)
- **/opt** On some Unix-like systems, this directory is used to store optional / additional software installed by the user
- **/proc** This directory contains the virtual (not real) proc file system, which is used to access live kernel parameters
- **/root** This directory is the home directory of the root user
- **/sbin** this directory contains the system programs useful to system administrators
- **/tmp** This directory contains a virtual file system that exists in RAM. It loses its contents upon reboot. Many software programs use this directory for creating temporary files
- **/usr** This directory holds the additional software installed by the user. It has a substructure that mirrors the structure of / (the root directory) to some extent
  - **/usr/bin** This subdirectory contains the executable programs for user-installed software
  - **/usr/include** This directory contains the header files included in C/C++ programs
  - **/usr/lib** This directory contains the libraries for the user-installed programs
  - **/usr/local** This directory contains programs for local use only (as opposed to use from a remote terminal). It also has a substructure similar to /usr
  - **/usr/man** This directory contains the pages of the online manual
  - **/usr/share** This directory contains shared files belonging to different programs
- **/var** This directory contains frequently changing data, including cache files, printer spool files as well as log files
  - **/var/log** This directory contains the various system and software logs. these logs are useful for debugging problems and to audit the system security
  - **/var/www/html** or **/var/www** this directory is the Apache web server web-root and only exists id Apache is installed

The Shells

Unix offers a variety of shells in both text mode and graphical mode. The graphical mode shells include the GNOME shell, Unity, KDE, XFCE, LXDE, MATE, etc. The commonly used text mode shells include sh (the original Bourne shell), ksh (Korn shell) csh (C Shell), bash (Bourne-Again SHell), etc.

The Graphical Shell (GNOME Shell / Unity)

Switching between programs

- ALT-TAB to switch between programs
- ALT-` (back-quote) to switch between windows of the same program

Multiple screens

- 12 virtual screens (6 text mode screens and 6 graphics mode screens) accessed using the shortcut keys CTRL-ALT-F1 ... CTRL-ALT-F12
- Each graphical screen can be configured to have multiple workspaces. Workspaces are arranged in a rectangular grid with some number of rows and some number of columns. To switch between workspaces and to drag and drop windows between workspaces, one may use the *Workspace Switcher*. Shortcut keys for switching between the workspaces are CTRL-ALT-Arrow Keys. To move a window to a different workspace, press ALT-SPACE to open the window menu and select appropriate option; or open *Workspace Switcher* and use drag and drop
- In the graphical mode, we may open one or more windows of the GNOME Terminal program to work in text mode. In each terminal window, we may have one or more tabs. The shortcut key for creating a new tab is CTRL-SHIFT-T and the shortcut keys for switching between tabs are CTL-PgUp and CTL-PgDown

The Bourne Shell

- **Internal commands** are built into the shell itself. There is no seperate executable file for them
- **External commands** are not built into the shell. There is a seperate executable file for them
- Search path
  - Since it is not practical to search the entire hard disk for the executable file corresponding to a possible external command entered by the user, only the directories in the search path are searched for the same. The search path is

stored in the built-in shell variable PATH as string containing one or more directories seperated by the : (colon) character
    - The current directory is not searched for a command if it is not in the search path
- CTRL-D at the beginning of a line indicates end-of-file (or end-of-input)
- CTRL-C is used to terminate the currently rnning foreground process
- CTRL-S is used to pause the terminal
- CTRL-Q is used to resume the terminal
- To copy text from the terminal, select the text using the mouse and press CTRL-SHIFT-C
- To paste text into the termnal, press CTRL-SHIFT-V. The text is inserted as if it had been typed by the user
- The shell maintains the history of the commands entered earlier in memory. When the shell session terminates, this command history is saved in the file ~/.bash_history. The command history can be accessed using the UP / DOWN arrow keys at the prompt. It is also available in subsequent sessions and survives reboots
- The shell has its own full-fledged built-in programming language with variables, control structures, input, output, etc. It is a dynamic language in nature
- Case sensitivity
- Command completion using TAB and TAB-TAB

The Command Line

- components of the command line and word splitting: The command line contains one (or more) command(s), arguments to the command, operators, etc. The shell performs word splitting on the command line using white space (space / tab characters) as the seperators
- In general, order of compnents of a command line is not important. However, there are several exceptions
- Options
    - **Short options** Short options consist of - (hyphen) followed by a single character
    - **Long options** Long options are specified using -- (two consecutive hyphens)
    - **Options with arguments** Some options have their own arguments. in such cases, the argument(s) must immediately follow the option
    - **Combining short options** Multiple short options can be combined by writing them immediately after a single hyphen: `ls -lh` . Only the last short option can have argument(s) in such case; and the argument, if any, is immediately specified after the combined options: `tar -czf backup.tar.gz directory`
- **Quoting and escaping** Strings on the command line can be specified without using any quotes. However, if a string contains white space or some other character having a special meaning to the shell interpreter, the string must be enclosed in quotes or

the white spaces / special characters must be escaped
    - **Escaping** A character having special meaning to the shell can be escaped by preceeding it with \ (backspace). This removes its special meaning. The special meaning of the backslash itself can also be removed by preceeding it with another backslash `\\`
    - **Single Quotes** Almost no pocessing is carried out inside strings enclosed in singl quotes `'aaa bbb ccc'`
    - **Double Quotes** Double quotes also behave like single quotes, but variable / parameter expansion *is* carried out inside double quotes: `$variable_name` is converted into the value of the variable. Hence if the value of variable *name* is *Scotty*, `echo 'Beam me up, $name'` outputs `Beam me up, $name` but `echo "Beam me up, $name"` outputs `Beam me up, Scotty`
- Operators (Input/Output Redirection, Combining Multiple Commands using Logical Operators)

File System Manipulation Commands

Note: *File system object* means either a file or a directory

- **pwd** (present working directory)
    - **pwd** Outputs the current working directory
- **cd** (change directory)
    - **cd directory** Changes the current directory to the given directory
    - **cd** changes the current directory to the home directory of the user. This behaviour is different from Windows where `cd` displays the current directory
- **ls** (list)
    - **ls** Outputs the names of the file system objects in the current directory
    - **ls arguments** Outputs the information about the argument(s). In case an argument is a file, its name is displayed. In case an argument is a directory, the names of file system objects inside the directory are displayed. If the argument file system object does not exist, an error message is output
    - **Options**
        - **-l** (*long* listing) Outputs a long listing with more information about the file system objects
        - **-d** (*directory*) When an argument is a directory, outputs information about the directory itself rather than its contents
        - **-R** (*recursive*) Outputs information about each argument along with child file system objects recursively
- **mkdir** (make directory)
    - **mkdir directory** Creates the given directory if it does not exist

- **rmdir** (remove directory)
  - Removes (deletes) the given directory if it is empty
- **cat** (concatenate)
  - **cat file(s)** Outputs a concatenation of the given files. Also used to output a single file
- **cp** (copy)
  - **cp arguments** There must be at least two arguments. The last argument is the target. All penultimate arguments are sources. If the target is a directory, the sources are copied into the target. If the target is a file, there must be only one source and the source file is copied onto the target file. If the target does not exist, it is assumed to be a file and is created. *cp* does not copy directories by default. There are no warning or prompts if a file is going to be overwritten
  - **Options**
    - **-r** (recursive) This option is used to copy an entire file system tree rooted at the given node (i.e. a directory along with all its contents recursively)
    - **-i** (interactive) Prompts before overwriting a file
- **rm** (remove)
  - **rm arguments** Removes (deletes) the given arguments. While the files and directories deleted from the GUI go into the trash, files and directories deleted from the command prompt are permanently lost. Does not delete directories by default. Does not prompt before deleting a file
  - **Options**
    - **-r** (recusrive) Deletes the arguments reursively. Directories in the arguments are deleted with all the contents recursively
    - **-i** (interactive) Prompts before deleting a file
- **mv** (move)
  - **mv arguments** Moves or renames as per the number and types of arguments
  - **How is the target of the move is determined**
    - If the destination is the name of an existing file, the target of move is assumed to be that file
    - If the destination is the name of an existing directory, the target is assumed to an object with the same name as the source inside that directory
    - If the destination does not exist, the destination name itself is the target. It is considered a file name if the source is a file and a directory name if the source is a directory
  - **When a move operation is performed**
    - If the directory containing the source and the directory containing the target are different, the source object is moved
  - **When a rename operation is performed**
    - If the source name and target name are not same, a rename operation is

performed

- **When a file is overwritten**
  - If the target is a file and it already exists, it is overwritten
- **Operations that are not allowed**
  - If there are more than one sources and the destination is not a directory
  - If the source is a directory and the target is an existing file
  - If the source is a filename and the target refers to the same file
- **Options**
  - **-i** (interactive) Prompts before overwriting a file

- **dirs** (directories)
  - **dirs** Displays the current directory stack
- **pushd** (push directory)
  - **pushd directory** Pushes the directory onto the directory stack and then changes to that directory
- **popd** (pop directory)
  - **popd** Pops the top-of-the-stack directory from the dirctory stack and changes to the new top-of-the-stack directory

Shell Globbing Patterns

Pattern	Matches with
?	Any single character
*	Any number of any characters
[abcde]	Any single character from among a, b, c, d and e
[^abcde]	Any one character other than a, b, c, d and e
[aeiou]	Any single vowel
[^aeiou]	Any one character other than a vowel
[abc^de]	Any one character from among a, b, c, ^, d and e
[a-z]	Any single lower case alphabetic character
[a-zA-Z]	Any single lower or upper case alphabetic character
[a-zA-Z_%]	Any single character from among alphabetic characters, _ and %
[a-zA-Z0-9]	Any single alphanumeric character
[^a-zA-Z0-9]	Any single non-alphanumeric character

Examples

Pattern	Matches with
p*	Anything starting with p
p*e	Anything that starts with p and ends with e, including pe
*~	All backup files
b??h	b, followed by exactly two characters, followed by h
b*h	Anything that starts with b and ends with h, including bh
dir*	Anything that starts with dir, including dir
dir?	dir followed by exactly one character

Plain Text Editors

- Plain text editors
  - Graphical editors
    - gedit / Text Editor (default)
    - gvim
    - leafpad
  - Text mode editors
    - vi (default)
    - vim (VI iMproved)
    - nano, pico
    - emacs (Editing MACroS)

Brief Overview of Working with the Vim Editor

- vim stands for VI iMproved
- It is an improved version of the vi editor
- vim has several *modes*
  - *Normal* (*command*) mode
    - When vim starts, it is in this mode
    - In this mode, the keys pressed by the user are interpreted as commands and corresponding actions are taken
  - *Insertion* mode
    - This mode is entered by pressing the *i* command in the command mode.

Several other commands also switch to insertion mode
• in this mode, the keys pressed by the user are treated as characters to be inserted into the text being edited at the curent cursor position
• One may switch back to the command mode by pressing `ESC` in the insertion mode

- *Last line* mode
  • This mode is entered when using certain commands in the command mode, like `:`
  • It is used for *ex commands* and long commands
  • The cursor moves to the last line where the rest of the command is entered
  • The command is completed (and executed) by pressing `ENTER` or `ESC`
  • To cancel the command, press `CTRL+C`

- Cursor movement keys like the arrow keys, HOME, END, PgUp, PgDn, etc. work as expected in both the command mode and the insertion mode in vim
- When vim is installed, the command *vi* also invokes *vim* only
- A file is opened by passing its name on the command line

```
vi filename
```

- A file is saved by typing `:w` (write) in the command mode
- To save the current file and close vi, type `:wq` (write and quit) in the command mode
- To close vi without saving the current file, type `:q!` (quit, with force) in the command mode
- An entire line can be deleted by pressing `dd` in the command mode
- *n* lines can be deleted by pressing `ndd` in the command mode. Here *n* is an integer and is known as the repeat count
- Deleting line(s) actually *cuts* them, so they can be pasted immediately afterwards
- `yy` (yank = copy) copies a line of text
- *n* lines can be copied by pressing `nyy` in the command mode.
- Text that is copied or cut (deleted) can be *put* (pasted) after the cursor using the command `p` (put) and before the cursor using the command `P`
- `u` (undo) can be used to undo the operations, while `CTRL-R` (redo) can be used to redo the operations that were previously undone

The Shell Scripting Language of the *bash* Shell

Shell Variables

- The shell variables need not be declared. A variable is created automatically the first

time it is assigned a value. All variables are of type *string*; but, in some contexts, may be interpreted as numbers. The variable assignment takes the form

```
variable_name=value
```

There must not be spaces around the = (assignment operator)

- The value of a variable can be accessed in the following ways:

```
$variable_name
${variable_name}
```

- **Built-in variables of the shell** The shell has a number of built-in variables. By convention, their names are in ALL_CAPS. For example, PATH, HOME, SHELL, TERM, etc.
- It is ***not*** an error to attempt to access the value of an undefined variable. Such an attempt simply returns an empty string (a string of zero length)
  - **SHELL** : The current shell
  - **HOME** : Home directory of the currently logged in user
  - **PATH** : A : (clolon) separated list of directories that are searched for external commands
  - **TERM** : The type of the current terminal
  - **PS1** : The primary prompt
  - **PS2** : The secondary prompt
- **Special parameters** The shell has several special parameters
  - **# ($#)**
  - *** and @ ($*, "$*", $@, "$@")**
  - **$?**
- **Debugging the shell script**
  - Use `set -xv` to turn debugging on
  - Use `set +xv` to turn debugging off

The `test` Command and the `[` Builtin

- The test command
  - The command `test` tests some condition and returns exit status accordingly (0 means success / true and non-zero means failure / false). It does not produce any output
  - The spacial parameter `?` can be used to output the exit status of the last foreground process

    - The syntax `[ condition ]` can be used in place of test condition (`[` is a shell builtin)

```
echo $?
```

- String tests
    - **-z *string*** Returns true if *string* has zero length (empty string)
    - **-n *string*** Returns true if *string* has non-zero length
    - ***string1* = *string2*** Returns true if *string1* is equal to *string2*
    - ***string1* != *string2*** Returns true if *string1* is not equal to *string2*
- Integer tests
    - ***no1* -gt *no2*** Returns true if *no1* is greater than *no2*
    - ***no1* -ge *no2*** Returns true if *no1* is greater than or equal to *no2*
    - ***no1* -lt *no2*** Returns true if *no1* is less than *no2*
    - ***no1* -le *no2*** Returns true if *no1* is less than or equal to *no2*
    - ***no1* -eq *no2*** Returns true if *no1* is equal to *no2*
    - ***no1* -ne *no2*** Returns true if *no1* is not equal to *no2*
- Logical operations on tests
    - ***test1* -a *test2*** (and) Returns true if both the tests evaluate to true
    - ***test1* -o *test2*** (or) Returns true if any one of the tests evaluates to true
    - ***!condition*** (not) Negation / logical inversion of condition
- New line in syntax
    - A `;` (semicolon) can be used wherever the syntax requires a new line

The Control Structures

- The *if* statement

```
if command1
then
    statement1
    statement2
    ...
elif command2
then
    statement3
    statement4
    ...
elif command3
then
    statement5
    statement6
```

```
    ...
 ...
 ...
else
    statement7
    statement8
    ...
fi
```

```
if test condition1
then
    statement1
    statement2
    ...
elif test condition2
then
    statement3
    statement4
    ...
elif test condition3
then
    statement5
    statement6
    ...
 ...
 ...
else
    statement7
    statement8
    ...
fi
```

```
if [ condition1 ]
then
    statement1
    statement2
    ...
elif [ condition2 ]
then
    statement3
    statement4
    ...
elif [ condition3 ]
then
    statement5
    statement6
    ...
 ...
```

```
 ...
else
    statement7
    statement8
    ...
fi
```

- The *while* statement

```
while command
do
    statement(s)
done
```

```
while test condition
do
    statement(s)
done
```

```
while [ condition ]
do
    statement(s)
done
```

- The *for* statement

```
for variable in white-space-separated-list
do
    statement(s)
done
```

- The *C-style for* statement

```
for ((initializer; test; increment))
do
    statement(s)
done
```

```
for ((i=0;i<10;i++))
do
    echo $i
done
```

- break and continue
  - `break` can be used in the loops to terminate the loop and jump to the next statement after the loop
  - `continue` can be used to terminate the current iteration of the loop and jump to the start of the loop
- The command `true` does nothing, but returns with an exit status of zero (success / true)
- The command `false` does nothing, but returns with a non-zero exit status (failure / false)

```
i=0
while true
do
    echo $i
    let i++
    if [ $i -ge 10 ]
    then
        break
    fi
done
```

- The *case* statement
  - cases are matched in sequence, and only the first matching case is executed
  - All cases except the last one must end with ;;
  - Multiple cases may be combined using | (the vertical bar character)
  - Shell globbing patterns may be used in cases
  - The pattern `*` stands for the default case. it must be the last case
  - The case statement ends with `esac`, reverse of *case*

```
read -p "Enter your option: " option
case $option in
1|4) echo one or four
     echo "(any one of them)"
   ;;
2) echo two
   ;;
3) echo three
   ;;
??) echo two characters
   ;;
a*) echo begins with an a
    ;;
*) echo other
esac
```

Comments

- The # character is used for writing single-line comments
- All the characters starting from the # upto the end of the line are treated as comment and are ignored by the shell interpreter
- There is no syntax for multi-line comments

The *Shebang* Line

- Unix-like systems offer several shells as well as interpreters for many different programming languages. The scripts can be executed as stand-alone commands (without mentioning the shell or the interpreter) as long as the shell or interpreter is present in the path. When a script is executed as a command, the shell tries to guess the interpreter to be used for its execution based on the contents, because the concept of file extensions is not strictly enforced in unix-like systems. Sometimes, such guess may be wrong also. To inform the shell which interpreter to use for executing a script in an explicit way within the shell script itself, a *shebang* line is used
- The *shebang* line is a special comment line mentioning the interpreter to be used for executing the script

```
#!/bin/bash
```

```
#!/usr/bin/perl
```

```
#!/usr/bin/python
```

- The *shebang* line must be the first line in a script

Output

- *echo*
  - `echo` is used to output its arguments
  - By default, echo appends a new line at the end of its output
  - To suppress that new line, use the `-n` option:
- *printf*
  - `printf` is used for formatted output
  - It syntax is identical to the printf function in the standard C library, but it is a

command
    - It does not append a new line at the end of its output by default

```
printf "%03d %20s %5.2f\n" $no $name $marks
```

Input

- *read*
    - The `read` shell builtin can be used to read the values of one or more variables
    - `read x` reads a line from the input and assigns it to the variable x
    - `read x y z` reads a line from the input, separates the words in the line and assigns them in order to the variables
    - If there are less words than variables, the remaining variables are not assigned
    - If there are more words than variables; all but the last variable are assigned one word while the last variable is assigned the rest of the line
    - The `-p` option can be used to display a prompt for reading

```
read -p "Enter a number: " no
```

Integer Arithmetic

- *expr*
    - The `expr` command can be used to evaluate complex expressions involving *integers*
    - It outputs the result of the evaluation
    - Usual operators are available
    - All the operators and arguments *must* be separated by white spaces
    - Rules of precedence and associativity are followed
    - Brackets can be used to group subexpressions
    - Charactes having a special meaning in the shell (like `(` and `*`) must be either quoted or escaped
    - The output of `expr` can be assigned to a variable using Command Substitution

```
expr 10 + 20
expr $x + $y
expr $x + $y \* $z
expr \( $x + $y \) \* $z
expr "(" $x + $y ")" "*" $z
expr 10 + \( $y - $x - \( $x + $y - \( 5 \* 3 \) \) \) \* 3
```

```
x=`expr $y + $z`
x=$(expr $y + $z)
```

- *$(( ))*
  - `$(( expression ))` can be used to evaluate `expression`
  - It is a form of Command Substitution. I.e., `$(( expression ))` in the command line is replaced by the result of evaluating `expression`
  - Usual operators are available
  - Operators and operands within `$(( ... ))` do not require spaces around them
  - Characters with special meaning in the shell (like `(` and `*`) need not be escaped or quoted
  - Values of variables may be accessed using the syntax `$variable` or just `variable`

```
x=$(( 10*20 ))
x=$(( (10+20)*20 ))
x=$(( ($x+$y)*$z ))
x=$(( (x+y)*z ))
x=$(( 10+(y-x-(x+y-(5*3)))*3 ))
x=$(( y++ ))
```

- *let*
  - `let` is a shell builtin for integer aritmetic
  - `let` expects an expression as a single argument (so the expression must not have spaces in it; or it should be quoted)
  - `let` evaluates its argument just like `$((...))
  - `let` is not a form of Command Substitution, so assignment must be used inside the let expression to store the calculated value in some variable

```
let x=10*20
let x=(10+20)*20
let x=y++
let i++
```

String Operations

- String concatenation
  - Two strings can be concatenated by simply placing them side-by-side without any intervening space

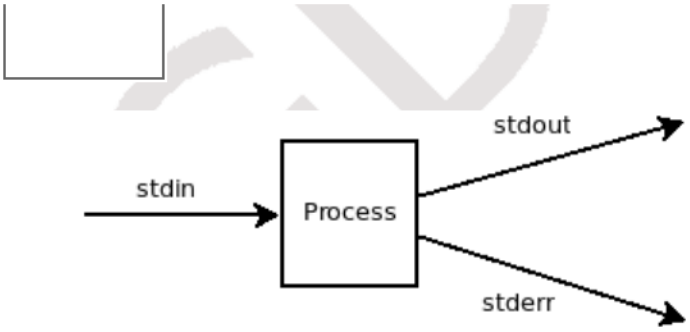
```
"abc""xyx"
$x$y
abc$x
${x}abc
```

- String length
  - Using `expr` : `expr length string` outputs the length of the string
  - Using bash parameter substitution: `${#variable}` results in the length of the variable
- Substring
  - Using `expr` : `expr substr str pos len` outputs a substring of maximum length *len* from the string *str*, starting at position *pos* (the first character has the position 1)
  - Using bash parameter substitution: `${variable:off:len}` returns a substring of maximum length *len* from the value of *variable*, starting from offset *off* (the first character has the offset 0)
- String matching
  - Using `expr` : `expr match str regexp` performs an anchored match of the string *str* with the regular expression *regexp*

Standard I/O Streams and I/O Redirection

Every process has a PCB (Process Control Block) associated with it. One of the items in the PCB is a table of open files. It lists all the files opened by the process. In UNIX systems, a file is opened using the *open()* system call. This system call returns an integer called the *file descriptor* (fd), which is the index of the entry of that file in the table of open files.

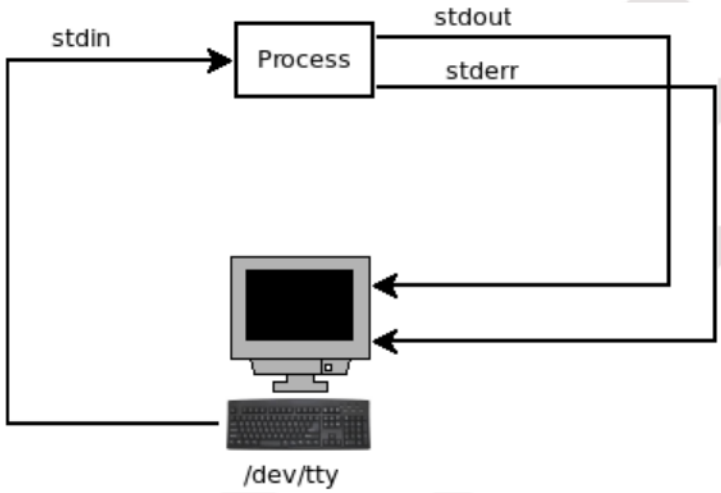
Table of Open Files
/dev/tty
/dev/tty
/dev/tty
f1
f2



Standard Console I/O Streams

Default Assignment of Standard I/O Streams

```
$ p1
```



Default Assignment of Standard I/O Streams

Table of Open Files for process p1

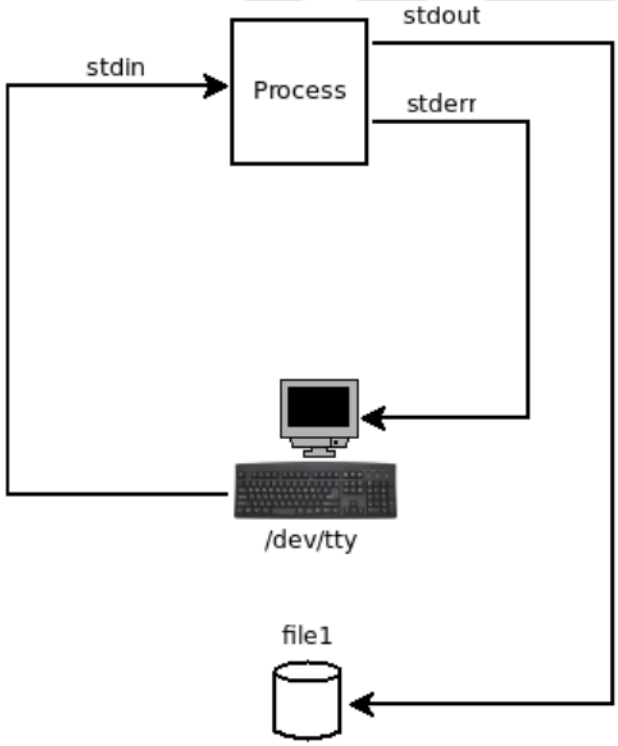
FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	/dev/tty
2	stderr	/dev/tty

Standard Output Redirection

```
$ p1 1>file1
```

```
$ p1 >file1

$ >file1 p1
```



Standard Output Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ file1
2	stderr	/dev/tty

The file *file1* will be opened in *write* mode by the shell *before* execution of the command. If it exists, it will be truncated.

```
$ >t1
```

This will create the file t1, if it does not exist; and will truncate the file t1, if it does.

```
$ xyz >file1     #there is no command called xyz in the path
bash: xyz: command not found
```

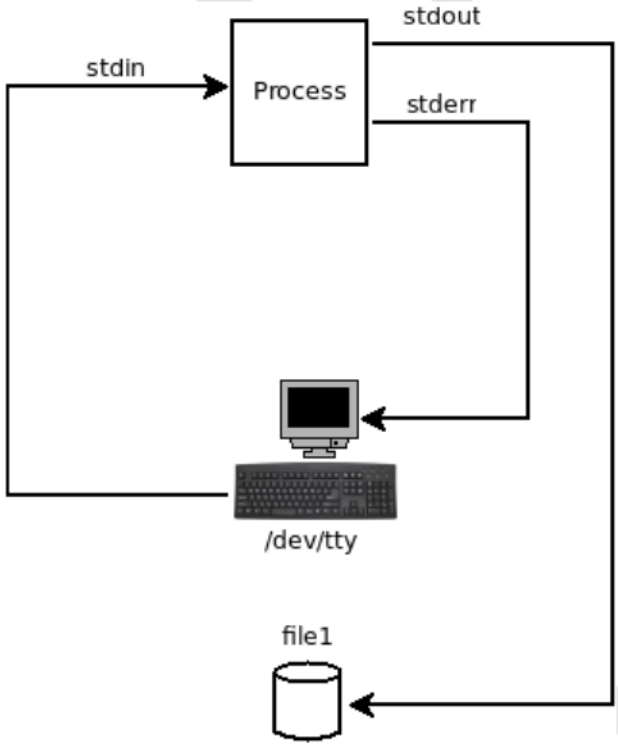
But the file file1 is still overwritten (truncated in this case).

Standard Output Redirection (Append Mode):

```
$ p1 1>>file1

$ p1 >>file1

$ >>file1 p1
```



Standard Output Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ file1
2	stderr	/dev/tty

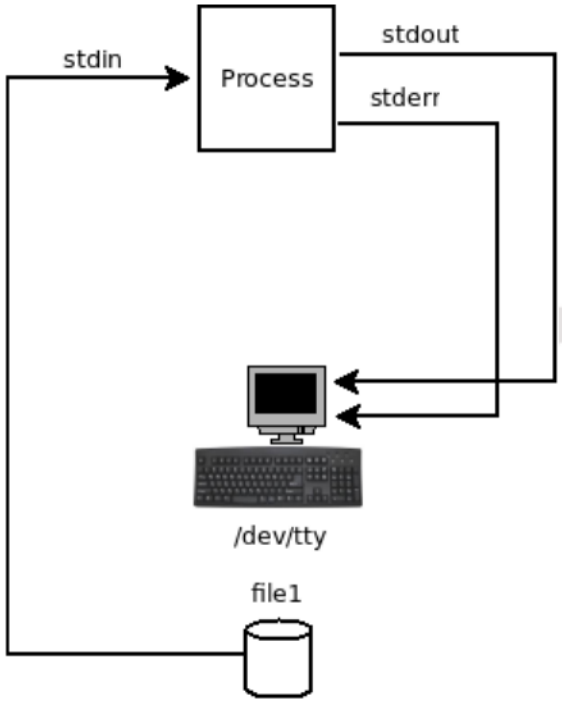
The file *file1* will be opened in *append* mode by the shell *before* execution of the command.

Standard Input Redirection

```
$ p1 0<file1

$ p1 <file1

$ <file1 p1
```



Standard Input Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	~~/dev/tty~~ file1
1	stdout	/dev/tty
2	stderr	/dev/tty

The file *file1* will be opened in *read* mode by the shell *before* execution of the command. If the file does not exist, it is an error.

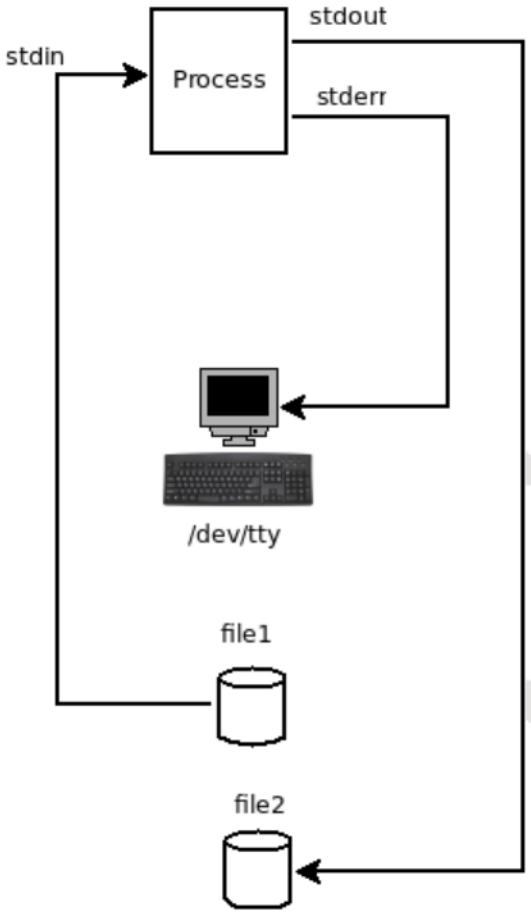
Standard Input and Standard Output Redirection

```
$ p1 0<file1 1>file2

$ p1 <file1 >file2

$ p1 >file2 <file1

$ >file2 <file1 p1
```



Standard Input and Standard Output Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	~~/dev/tty~~ file1
1	stdout	~~/dev/tty~~ file2
2	stderr	/dev/tty

FD	Name	Associated with

The file *file1* will be opened in *read* mode by the shell *before* execution of the command. If the file does not exist, it is an error.

The file *file2* will be opened in *write* mode by the shell *before* execution of the command. If it exists, it will be truncated.

file1 and *file2* must not be same files. If the following command is executed:

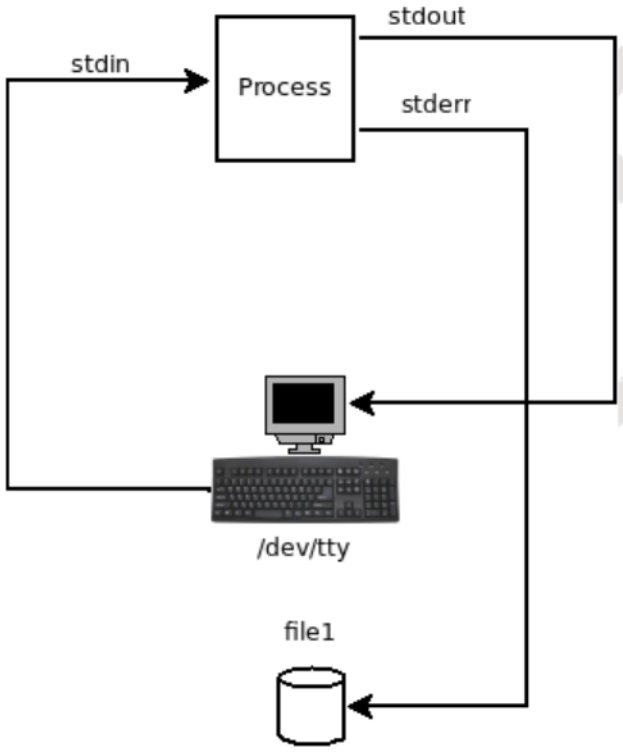
```
$ p1 <zzz >zzz
```

The file *zzz* will be overwritten *before* the execution begins and it will be an empty file.

Standard Error Redirection

```
$ p1 2>file1

$ 2>file1 p1
```



Standard Error Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	/dev/tty
2	stderr	~~/dev/tty~~ file1

The file *file1* will be opened in *write* mode by the shell *before* execution of the command. If it exists, it will be truncated.

If we use 2>> operator in place of 2> operator, the file will be opened in append mode.

```
$ p1 2>>file1
```

/dev/null is the null device. Any output sent to it is simply discarded. If we try to perform input from it, we immediately get EOF (End-of-File).

```
$ find / -name '*.php' 2>/dev/null
```

Supplying the standard input from the command line

```
$ p1 << END
INPUT LINE 1
INPUT LINE 2
INPUT LINE 3
INPUT LINE 4
END
```

This way of providing input from the command line is known as *the here document*.

Command Substitution

Command substitution means constructing the command line by substituting a command in it by the standard output of that command.

```
$ p1 p1_arg1 p1_arg2 `p2 p2_arg1 p2_arg2 p2_arg3 ...` p1_argx p1_argy p1_argz
```

The command

```
p2 p2_arg1 p2_arg2 p2_arg3 …
```

will be executed as an independent command and the standard output from it will replace the whole backquoted string in the original command line. Then the modified command line will be executed.

E.g., if the standard output of

```
p2 p2_arg1 p2_arg2 p2_arg3 …
```

is

```
aaa bbb ccc
```

then the modified command line becomes

```
p1 p1_arg1 p1_arg2 aaa bbb ccc p1_argx p1_argy p1_argz
```

Which will then be executed.

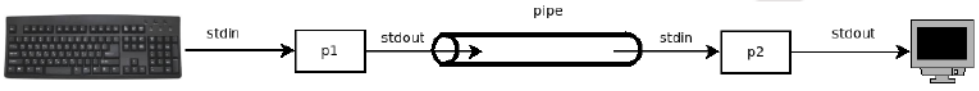
```
 x=10
y=20
z=`expr $x + $y`
+ expr 10 + 20
x=30
```

New Syntax

```
$ p1 p1_arg1 p1_arg2 $(p2 p2_arg1 p2_arg2 p2_arg3 ...) p1_argx p1_argy p1_argz
```

Pipes

```
$ p1 | p2
```



Pipe

Table of Open Files for process p1

FD	Name	Associated with

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ pipe1
2	stderr	/dev/tty

Table of Open Files for process p2

FD	Name	Associated with
0	stdin	~~/dev/tty~~ pipe1
1	stdout	/dev/tty
2	stderr	/dev/tty

The processes p1 and p2 will be run concurrently as child processes of the shell, with the standard output of p1 redirected to a pipe and p2 taking its standard input from that pipe.

```
$ p1 | p2 | p3 | p4 | p5
```

p1 will take its standard input from the terminal (keyboard). Standard output of p1 will be supplied as standard input to p2. Standard output of p2 will be supplied as standard input to p3. Standard output of p3 will be supplied as standard input to p4. Standard output of p4 will be supplied as standard input to p5. Standard output of p5 will appear on the terminal (monitor). In a chain of commands, each process carries out one specific stage of text processing.

Redirecting One File Descriptor to Another File Descriptor

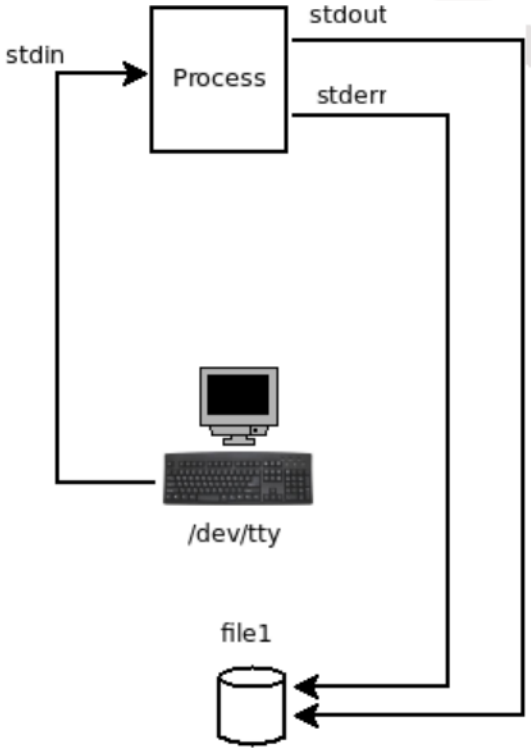
```
$ p1 >file1 2>file1
```

This will not have the desired effect.

```
$ p1 >file1 2>&1
```

This means send standard error wherever File Descriptor 1 (standard output) is going.

First the standard output will be redirected to file1 and then the standard error will be redirected to wherever the standard output is going, i.e. file1. Hence, both standard output and standard error will be redirected to file1.



Standard Output and Standard Error Redirection to the Same File

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ file1
2	stderr	~~/dev/tty~~ file1

In this case order of redirection *is* significant!

Suppose, instead of

```
$ p1 >file1 2>&1
```

we run

```
$ p1 2>&1 >file1
```

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ file1
2	stderr	/dev/tty

First the standard error will be redirected to wherever the standard output is going, i.e. the terminal and then the standard output will be redirected to file1.

Arguments for Different Redirection Operators

command1	>	file1
command1	>>	file1
Command1	2>	file1
command1	2>>	file1
command1	<	file1
command1	|	command2
command1	<<	WORD

Filters

A *filter* is a command that takes its input from standard input, processes it and produces the output on standard output. The benefit of a filter is that its input and output can be redirected independently and flexibly. Also, several filters can be chained in a pipeline.

```
cat a4 | grep '\^\[0-9\]\*,\[0-9\]\*,12'
```

Regular Expressions

Basic Regular Expressions

Pattern	Matches with
.	Any single character
*	Previous element 0 or more times
^	An imaginary anchor at the beginning of line (only when ^ is at the beginning of the pattern)
$	An imaginary anchor at the end of line (only when $ is at the end of the pattern)

\ is the *escape character* - it removes the special meaning of the next character; also adds special meaning to some characters in some circumstances as described below.

Character Classes

Pattern	Matches with
[abcd]	Any one character from a, b, c or d
[a-z]	Any one character between a and z, inclusive
[a-z0-9]	Any one character from between a and z, inclusive; or from between 0 and 9, inclusive
[a-zA-Z0-9]	Any one character from between a and z, inclusive; or from between A and Z, inclusive; or from between 0 and 9, inclusive
[a-z0-9ABCD]	Any one character from between a and z, inclusive or from between 0 and 9, inclusive or A or B or C or D
[^a-z0-9]	Any one character other than those between a and z, inclusive and those between 0 and 9, inclusive
[a-z^0-9]	Any one character from between a and z, inclusive or ^ or from between between 0 and 9, inclusive

Pattern	Matches with
\t	The TAB character
\n	The newline character
\\	The (backslash) character
\w	equivalent to [a-zA-Z0-9]
\W	equivalent to [^a-zA-Z0-9]

Predefined Character Classes

Pattern	Matches with	Equivalent to
[:alpha:]	Any alphabetic character	[a-zA-Z]
[:digit:]	Any digit	[0-9]
[:alnum:]	Any alphanumeric character	[a-zA-Z0-9]
[:lower:]	Any lower case letter	[a-z]
[:upper:]	Any upper case letter	[A-Z]
[:space:]	Any white space character	space/tab/new line
[:punct:]	Any punctuation character	
[:xdigit:]	Any hexadecimal digit	[0-9a-fA-F]
[:print:]	Any printable character	
[:cntrl:]	Any control character	
[:graph:]	Any ASCII graph character	

Word Matching

Pattern	Matches with
\<	An imaginary anchor at the beginning of every word
\>	An imaginary anchor at the end of every word
\<pat1\>	The input matches with the pattern \<pat1\> only if the input matches with pattern pat1 and is a whole word (not part of a larger word)

Extended Regular Expressions

The meta characters used in extended regular expressions are as follows. These meta characters can be used as given (individually) when extended regular expressions are enabled (e.g. grep -E). The grep command also supports them in basic regular expressions in their escaped form (e.g. \+ instead of +, \? instread of ?, \{ instread of {, etc.).

Pattern	Matches with
+	Previous element 1 or more times
?	Previous element 0 or 1 time

Repetition

{*m*} in a pattern means the previous element *m* times. {*m*,*n*} in a pattern means the previous element between *m* and *n* times. {*m*,} in a pattern means the previous element *m* or more times.

Notes:

- The input abc will match with the pattern .{3} because the pattern .{3} is treated as equivalent to the pattern ...
- The pattern matchers are usually greedy; they try to match the given pattern or part of the pattern with the maximum possible amount of input.

Alternation

pattern1 | pattern2 matches any input that matches with either pattern1 or pattern2

Subexpressions / Groups / Tags and Back References

Any part of the search pattern enclosed between (and) is treated as a subexpression / group (considered to have been tagged). The exact input that matches with a subexpression is remembered and can be referenced later as *n*, where *n* is the subexpression number (you can have multiple subexpressions in your search pattern and they are numbered serially starting from 1, so \1 means the input that matched with the first subexpression, \2 means the input that matched with the second subexpression and so on).

Examples:

Palindrome strings of 5 characters:

```
^\(.\)\(.\).\2\1$
```

Line containing only valid numbers (integers, floating point numbers in usual notation, numbers in scientific notation):

```
^[+-]\?[0-9]\+\(\.[0-9]*\)\?\([eE][-+]\?[0-9]\+\)\?$
```

The grep Family of Commands

- **grep:** regular grep, did not support multiple patterns and extended regular expressions
- **egrep:** extended grep, supported multiple patterns and extended regular expressions
- **fgrep:** fast grep, did not support patterns (can search only for fixed strings), but supported multiple search strings

Note: under GNU/Linux, the functionalities of all three commands have been combined into a single command - grep

grep -E is equivalent to egrep

grep -F is equivalent to fgrep

Feature	grep	egrep (extended)	fgrep (fast)
Regular Expressions	Yes	Yes	No
Extended Regular Expressions	No	Yes	No
Multiple Patterns	No	Yes	Yes
Speed	Medium	Medium	Fast

Combining Multiple Commands using Logical Operators

Note: An exit status of 0 means success, while any non-zero exit status means failure

- `cmd1 ; cmd2` cmd1 is executed and then cmd2 is executed (always)
- `cmd1 && cmd2` cmd1 is executed and then cmd2 is executed only if cmd1 is successful

- `cmd1 || cmd2` cmd1 is executed and then cmd2 is executed only if cmd1 is unsuccessful
- `(cmd1;cmd2)` the command list is executed in a child shell
- `{cmd1;cmd2;}` the command list is executed in the present shell

Vim Editor in More Detail

Commands List

- **File System Handling**
  - **pwd** Displays the present working directory
  - **ls** Displays list of files
    - **-a** Display all files, including hidden ones and the implied . and .. directories
    - **-A** Display all files, including hidden ones but excluding the implied . and .. directories
    - **-d** When an argument is a directory, display the directory (default is to display the contents of the directory)
    - **-h** Display human-readable size (20M, 1.5G, etc.)
    - **-i** Display the inode number also
    - **-l** Display a long list
    - **-r** Use reverse (descending) order while sorting
    - **-R** Display the contents of the arguments recursively (in case of a directory, display the subdirectories, subsubdirectories, etc.)
    - **-S** Sort list on file size
    - **-t** Sort list on modification time
    - **-X** Sort list on filename extension
    - **-1** Single-line output, even on a terminal
    - **--color=always|never|auto** Produce color output always/never/decide automatically based on the standard output destination
  - **cat** Output standard input / a file / concatenate multiple files and output
  - **cd** Change directory
  - **pushd** Push the current directory on the directory stack and change to the argument directory
  - **popd** Pop a directory from the directory stack and change to that directory
  - **dirs** Display the current directory stack
  - **mkdir** Make one or more directories
  - **rmdir** Remove one or more directories (must be empty)
  - **rm** Remove file(s) (and directories)

        - **-r** Remove file(s) and directories recursively
        - **-i** Remove interactively (prompt before deleting each file/directory)
        - **-f** Ignore non-existant files, never prompt
    - **cp** Copy file(s)
        - **-r** Copy file(s) and directories recursively
        - **-i** Copy interactively (prompt before deleting each file/directory)
    - **mv** Move/rename files and directories
        - **-i** Move/rename files and directories interactively (prompt before deleting each file/directory)
        - **-f** Move/rename forcefully, do not prompt before overwriting
    - **touch** Set the last modification time stamp of the argument file to current date & time, create the argument file, if it does not exist
    - **chmod** change the mode (permissions) of the file
        - **Symbolic syntax**
        - **Numeric syntax**
        - **Special permissions**
        - **-R** Change mode recursively
    - **chown** Change the owner and/or group of files and directories
        - **Changing only the owner**
        - **Changing both the owner and group**
        - **Changing only the group**
        - **-R** Change the owner and/or group recursively
    - **chgrp** Change the group of files and directories
        - **-R** Change the group recursively
    - **umask** Set the default file permissions mask
    - **ln** Create a link (hard link by default, symbolic link with the -s option)
        - **-s** Create a symbolic link
    - **mount** Mount a file system onto a directory
        - **-t** Specify the file system type
        - **-a** Mount all file systems from /etc/fstab having the auto option
        - **-o** Supply mount options (e.g. -o loop, -o rw)
        - **-r** Mount read only
        - **-w** Mount for read/write
    - **umount** Unmount a file system
- **Editing**
    - **vim**
- **Process Management**
    - **ps** process Status; display a list of running processes (by default, only processes in the current session)
        - **-a** (all) processes belonging to the current user sessions except the

session leaders (like the shell process that is the root of the process hierarchy for the session)

- **-e** (every) Displays all the interactive as well as non-interactive processes of all the users, including the system processes
- **-f** (full) Show the full information including parent process id and the full command line

- **kill** Send a signal to another process
  - **-9** Send the kill signal to another process to kill it

- **Basic Filters**
  - **cut** Extract one or more fields (columns) from input
    - **-c** Specify column positions (for use with fixed-width input)
    - **-d** Specify delimiter (for use with delimited input)
    - **-f** Specify fields to be extracted (for use with delimited input)
  - **grep (egrep, fgrep)**
    - **-E** Enable extended regular expression
    - **-e** Specify a pattern (used to specify multiple patterns)
    - **-F** Use fixed strings (work like fgrep)
    - **-f** Specify a file containing patterns, one per line
    - **-i** Ignore case when matching patterns
    - **-c** Output a count of matching lines
    - **-v** Invert match, output lines other than those having a match for the pattern(s)
    - **-w** word match, pattern must match with a whole word in the line
    - **-l** Output only a list of files that have one or more matches
    - **-n** Output line numbers along with matching lines
    - **--color=always|none|auto** Produce color output always/never/decide automatically based on the standard output destination
  - **pgrep** Search processes based on name / command line arguments
    - **-a** Display the full command line of the process that matched
    - **-f** Match the full command line rather than just the process name
  - **wc** Word Count; count the number of lines, words and characters in standard input or a file
    - **-l** Output line count
    - **-w** Output line count
    - **-c** Output line count
  - **head** Display the first few lines of a file or standard input (10 by default)
    - **-n** number of lines from the beginning to be displayed
  - **tail** Display the last few lines of a file or standard input (10 by default)
    - **-n** number of lines from the end to be displayed
    - **-f** continue to wait for further input even after reaching end of file; useful for

watching a continuously growing file
    - **paste** "Join" files side-by-side
    - **sort** Sort the input
        - **-k** Specify key (starting column, ending column) on which to sort. In case of delimited input, specify starting and ending field number, e.g. -k2,2. In case of fixed-width input, specify 0.starting character position and 0.ending character position, e.g. -k0.5,0.10 May be repeated for sorting on multiple keys
        - **-r** sort in reverse (descending) order
        - **-t** Specify the delimiter
        - **-d** Sort as per dictionary (lexical) order (default)
        - **-f** Fold lowercase to uppercase (ignore case while sorting)
        - **-n** Sort numerically
        - **-h** Sort numerically, human readable sizes (K, M, G, etc.)
    - **uniq** Output unique or repeated lines (assumes the input is sorted)
        - **-c** Prefix lines by a count of how many times the line is repeated
        - **-d** Only output duplicated (repeated) lines
        - **-i** ignore case
        - **-u** Only output unique lines (default)
    - **tr** Translate - convert or delete individual characters
        - **-s** Squeeze - convert multiple consecutive occurrences of the given characters into a single occurrence
        - **-d** Delete the given characters from the input
    - **tee** Copy standard input to given file as well as to standard output
        - **-a** append to file rather than overwrite
- Advanced Filters
    - **sed** Stream editor. Used to edit a file programmatically (from script)
        - **-n** Do not perform default output
        - **-f** Specify sed script filename
        - **commands**
            - **s** Substitute
            - **i** Insert line(s) before
            - **a** Append line(s) after
            - **d** delete line
            - **p** print (output) line
    - **awk** Powerful data manipulation program
        - **-f** Specify awk script filename
        - **Built-in Variables**
            - **ARGC** Count of command line arguments

- **ARGV** Argument vector (array of command line arguments)
- **ENVIRON** Array of environment variables
- **FILENAME** Name of the current file
- **FNR** File Number of Record (line number of the current line in the current file)
- **FS** Field separator
- **NF** Number of fields in the current line
- **NR** Number of Record (a cumulative line number for all the lines from all the files processed so far)
- **OFS** Output field separator (inserted between fields by print)

• **Built-in Functions** (Note: s/t → string, r → regular expression, n → number, i → index, a → array)

- **length(s)** length of string s
- **index(s, t)** 1-based index of substring t in s, 0 if t is not found in s
- **match(s, r)** returns the 1-based index of the first longest match of regular expression r in s, 0 otherwise
- **split(s, a, r)** Splits string s into fields using the separators in r (if omitted, default is white space) and stores them into array a as elements starting with index 1
- **sub(r, s, t)** Substitution. First match of r in t is replaced by s
- **gsub(r, s, t)** Global substitution. Every match of r in t is replaced by s
- **substr(s, i, n)** Return substring of s, starting at index i (1-based) and of length n. If n is omitted, returns a suffix of s starting at i
- **tolower(s)** Converts s to lower case
- **toupper(s)** Converts s to upper case
- **print(expr-list)** Print (output) each argument separated by OFS
- **printf(format, expr_list)** Just like the printf function in the C programming language
- **sprintf(format, expr-list)** Just like printf, but instead of performing output, returns a string containing the formatted output
- **atan2(y, x)** Arctan of y/x between -pi and pi
- **cos(x)** Cosine of x (x in radians)
- **exp(x)** exponential of x
- **int(x)** integer portion of x, truncated towards 0
- **log(x)** natural logarithm of x
- **rand()** returns a random number between 0 and 1
- **sin(x)** Sine of x (x in radians)
- **sqrt(x)** square root of x
- **getline var < filename** opens the file if it is not already open and reads the next line from it into variable var. If var is omitted, reads into $0 and

updates the fields and NF. If filename is omitted, the current file is used
› **close(filename)** closes the file
› **system(*cmd*)** runs the command *cmd* in a subshell and returns its exit status

- **Other Commands**
  - **bash** run a child shell, If a file name is passed as an argument, interpret it as a shell script and exit from the child shell
    - **-c** run the command c in a child shell and exit from the child shell
  - **clear** clear the screen
  - **more** displays a file or standard input one page at a time. Press ENTER for next line, SPACE for next page, b for previous page (not supported with pipes), q to quit, /patternENTER search, n/N to search again in forward/backward direction
  - **less** More powerful version of more, supports two-way scrolling with pipes as well
  - **echo** outputs its arguments followed by a newline
    - **-n** do not output a newline at the end
  - **read var1 var2 var3...** reads a line, performs word-splitting and assigns each word to corresponding variable. If there are less words than the number of variables then the remaining variables remain unassigned. If there are more words than the number of variables then one word is assigned to each variable except the last one; the last variable holds all remaining input
    - **-p** display the argument as a prompt if the input comes from a terminal
    - **-s** do not echo (display) the characters as they are read (useful for input of passwords)
  - **test** test various conditions
    - **string tests**
      - **-z *str*** True of the string *str* is zero length (empty string)
      - **-n *str*** True of the string *str* is non-zero length
      - ***str1* = *str2*** True if *str1* is equal to *str2* (string comparison)
      - ***str1* != *str2*** True if *str1* is not equal to *str2* (string comparison)
    - **integer tests**
      - ***no1* -gt *no2*** True if *no1* is greater than *no2*
      - ***no1* -ge *no2*** True if *no1* is greater than or equal to *no2*
      - ***no1* -lt *no2*** True if *no1* is less than *no2*
      - ***no1* -le *no2*** True if *no1* is less than or equal to *no2*
      - ***no1* -eq *no2*** True if *no1* is equal to *no2* (integer comparison)
      - ***no1* -ne *no2*** True if *no1* is not equal to *no2* (integer comparison)
    - **file related tests**
      - **-e *file1*** True if the file *file1* exists
      - **-d *file1*** True if the file *file1* exists and is a directory

        - **-f *file1*** True if the file *file1* exists and is a regular file
        - **-r *file1*** True if the file *file1* exists and is readable for the current user
        - **-w *file1*** True if the file *file1* exists and is writable for the current user
        - **-x *file1*** True if the file *file1* exists and is executable for the current user
        - **-s *file1*** True if the file *file1* exists and has a size greater than zero
    - **expr** evaluate expressions
        - **arithmetic expressions** expr
        - **string expressions** String Operations
    - **bc** arbitrary precision integer and floating arithmetic tool
    - **printf** Works just like the `printf()` function in the C programming language. As it is a command, its syntax is slightly different:
      `printf format-string arg1 arg2 arg3 ...`
    - **man** show the manual page for the argument from the first section in which it exists
        - **-a** show manual pages for the argument from all the sections in which it exists one after another
    - **info** more detailed on line manual (Linux only)
    - **whatis** Only display the heading of the manual page for the argument
    - **apropos** Search the headings of all manual pages for the given arguments and display the matching headings (useful for finding a command when one doesn't know the command name)
    - **which** display the executable corresponding to a command by searching PATH (e.g. which ls outputs /bin/ls), only works for external commands
    - **type** displays from where a command will be run by the shell, works for both internal and external commands
    - **file** Guess the type of content in a file by looking at some initial portion of the file content
    - **comm** display lines that are common to two files and lines that are not (files are assumed to be sorted)
        - **-1** do not display the first column (lines unique to the first file)
        - **-2** do not display the second column (lines unique to the second file)
        - **-3** do not display the third column (lines common to both the files)
        - **--check-order** check that the input is correctly sorted
        - **--nocheck-order** do not check that the input is correctly sorted
    - **diff** compare two files and output differences between them
        - **-i** ignore case while comparing
        - **-e** produce output compatible with the ed editor
    - **patch** apply a diff patch to the original file to convert it to the modified file (note: the original file is modified)
    - **join** join two files based on some common field(s), like a join in a relational

database

- **-a** print unpairable lines from file filenum (an argument, must be 1 or 2)
- **-e** replace missing input fields with empty strings
- **-i** ignore case when comparing fields for join condition
- **-t** specify field separators
- **-1** specify the field number of the join field in the first file
- **-2** specify the field number of the join field in the second file

- **who** display a list of currently logged-in users
  - **-a** display all information about currently logged-in users
  - **am i** display information about who is the logged-in user in the current session
- **whoami** display only the name of who is the logged-in user in the current session
- **date** get/set the system date in a variety of formats, convert dates between formats, extract components from dates
  - **-d** Use the given date instead of the system date
  - **-s** set the system date
  - **formats**
    - **%a** abbreviated week name (Sun..Sat)
    - **%A** full week name (Sunday..Saturday)
    - **%b** abbreviated month name (Jan..Dec)
    - **%B** full month name (January..December)
    - **%d** day of the month (01..31)
    - **%Y** Year in four digits
    - **%H** hour in 24-hour format (00..23)
    - **%I** hour in 12-hour format (01..12)
    - **%m** month number (01..12)
    - **%M** minute (00..59)
    - **%p** AM/PM as applicable
    - **%P** am/pm as applicable
    - **%s** number of seconds since 1970-01-01 00:00:00 UTC
    - **%S** seconds (00..60)
    - **%u** day of the week as a number (1..7, 1 is Monday)
- **cal** Display the calender of the given month and year (year must be 4-digit)
- **find** find files in the given directory recursively matching given criteria
  - **Predicates**
    - **-name *fname*** match filename with *fname*
    - **-iname *fname*** match filename with *fname*, case insensitive
    - **-path *p*** match a part of the file's path, including filename with *p*
    - **-type *t*** match files having type *t*, *t* may be *f* for regular file, *d* for

directory
- **Numeric arguments**
  - ***n*** means exactly equal to *n*
  - ***+n*** means >= *n*
  - ***-n*** means <= *n*
- **-mmin *n*** files modified *n* minutes ago
- **-mtime *n*** files modified *n* days ago
- **-empty** match empty (zero-length) files
- **-size *n*** match files having size *n*
- **-user *u*** match the file's owner with *u*
- **-group *g*** match the file's group with *g*
- **-readable** match files on which the current user has read permission
- **-writable** match files on which the current user has write permission
- **-executable** match files on which the current user has execute permission
- **-perm *p*** files having symbolic or numeric permissions *p*

- **actions**
  - **-delete** delete the matching file
  - **-exec** execute the given command on the matching file (must be terminated with \;)
  - **-ok** same as -exec, but prompt for confirmation before executing the command
  - **-print** display the filename with path (default action)
- **Operators**
  - **-a** and
  - **-o** or
  - **!** not
  - **()** Brackets can be used for grouping

- **tty** display the filename of the current terminal
- **tar** tape archiver. Combines several files into a single archive file (but does not compress it by default) on tape/disk
  - **-c** create an archive
  - **-x** extract from the archive
  - **-t** list from the archive
  - **-f** specify the filename of the archive
  - **-v** verbose mode, display each file as it is processed
  - **-z** compress/uncompress the file using the zip algorithm
- **gzip** replace each file by its compressed version (using the zip algorithm) and add extension .gz, can only compress individual files
- **gunzip** replace each file by its uncompressed version (using the zip algorithm)

and remove extension .gz/.Z, can only uncompress individual files

- **dd** Disk Dump. Used to read/write storage devices directly. Can be used to create and restore disk images
- **eval** The arguments are executed by the shell as a single command and its exit status is returned. Useful for executing dynamically generated shell commands
- **df** show free disk space on all mounted file systems. By default usage is shown in terms of 1024 byte blocks
    - **-h** show human readable figures (200M, 2.3G, etc.)
- **du** show the amount of disk space used by the argument files/directories. In case of directories, show total space used by the directory and its contents (calculated recursively). By default usage is shown in terms of 1024 byte blocks
    - **-s** show only summary (total) for each argument
    - **-h** show human readable figures (200M, 2.3G, etc.).
- **hd** Hex Dump. Used to view the contents of binary files
- **hostname** Output the name of the computer system
- **uname** Output system information
- **sudo** Execute the given command as a superuser. The currently logged-in user must have administrative privileges. Prompts for the current user's password for the first time, remembers the authorization for 15 minutes or until the session terminates.
    - **-k** Forget authorization, will prompt for password next time sudo is executed
    - **-s** Run a shell with administrative privileges
- **gksudo** Run a graphical program with administrative privileges (e.g. gksudo nautilus /)