Introduction to Linux

Training Material

License

Except where otherwise noted, this work is licensed under
http://creativecommons.org/licenses/by-nc-sa/3.0/

Atlantic LINUX

Contents

Atlantic LINUX

Contents

Atlantic LINUX

Contents

Atlantic LINUX

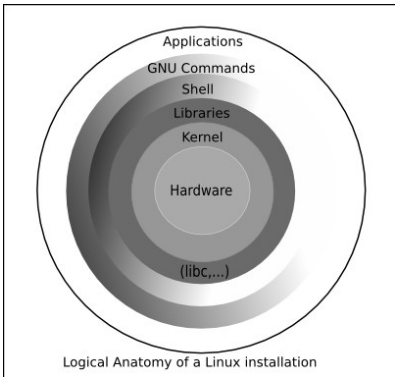
Contents

Atlantic LINUX

Introduction

What is Linux?

- Kernel
  - 2.4.x
  - 2.6.x
- Distributions
  - Red Hat
  - Novell / SuSE
  - Debian
- GNU
  - compilers
  - libraries
  - editors and other tools



Logical Anatomy of a Linux installation

Atlantic LINUX

- Linux is a free operating system (although some distributions do provide a traditional licensing model).
- Linux provides a secure platform – including a kernel firewall, a secure use model (users, groups, password-based authorisation) and support for advanced security models (LinuxSE and capabilities).
- There is a lot of high quality software (both open source and commercial) available for the Linux operating system from database software (including Oracle RDBMS) to application software (including Sun's OpenOffice).
- The Linux kernel is highly-portable (currently available on 14 different architectures ranging from big-iron such IBM's S390 to common platforms such as x86 and embedded platforms such as SuperH).
- Linux is supported by large ISVs (Oracle, IBM and SAP) and leading vendors (IBM, Dell, HP).

A brief history of Linux

Sep 1983 – Richard Stallman announces the GNU Project

Apr 1991 – Linus Torvalds announces he's working on a hobby OS

Sep 1991 – Linux Kernel 0.01

Mar 1994 – Linux Kernel 1.0 (i386)

Mar 1995 – Linux Kernel 1.2 (Alpha, Mips, Sparc)

June 1996 – Linux Kernel 2.0 (SMP, Tux the Penguin)

Jan 1999 – Linux Kernel 2.2 (64-bit, FAT32, NTFS)

Jan 2001 – Linux Kernel 2.4 (ISA PnP, PA-RISC, USB, PC Card)

Dec 2003 – Linux Kernel 2.6 (IA64, x86_64, em64t, embedded systems, NUMA)

Atlantic LINUX

From: Linus Benedict Torvalds (torvalds@klaava.Helsinki.FI)
Subject: What would you like to see most in minix?

Newsgroups: comp.os.minix
Date: 1991-08-25 23:12:08 PST

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

Linux distributions (1/2)

- Red Hat
  - Enterprise Linux (AS, ES, WS, Desktop) v4
  - Enterprise Linux (AS, ES, WS) v3
  - Fedora Core
- Debian
  - Stable (3.1 - *Sarge*)
  - Testing (*Etch*)
  - Unstable (*Sid*)
  - Derivatives (Ubuntu, Xandros, Knoppix, Progeny)

Red Hat (http://www.redhat.com/)

RHEL v4 - release Feburary 2005 includes 2.6 kernel, ext3 qand lvm updates, security enhanced linux compiler updates, gnome 2.8

RHEL v3 - introduced in September 2003

Enterprise Linux software has much longer support cycle (7 years)

Fedora is the proving ground for RHEL releases (effectively RHEL beta), available for free use but not supported.

Debian (http://www.debian.org/)

The debian distributions are non-commerical. a volunteer project consisting of about 1000 active developers packaging others peoples software and integrating it into a new distribution. Debian makes a release every few years which they label stable. Debian's key feature is that it can be installed and upgraded over the network with a very powerful tool for managing dependencies. It is thus possible to track the current versions of Debian in development (testing and unstable).

Debian is used as a base for a number of other Linux Distributions (both commercial and non-commercial). The most notable commercial ones are Ubuntu (http://www.ubuntu.com/) and Xandros (http://www.xandros.com/). The Knoppix distribution used for system recovery, which runs directly from the CD is also derived from Debian.

Linux distributions (2/2)

- Novell / SuSE
  - SuSE Linux Enterprise Server 10
  - SuSE Linux Enterprise Server 9 (SP3)
  - SuSE Linux Enterprise Desktop 10
  - Novell Linux Desktop 9
  - OpenSuSE 10.1
- Others
  - Gentoo
  - Mandriva
  - Ubuntu
  - Sun Java Desktop System
  - Slackware
  - Turbolinux

Atlantic LINUX

SuSE / Novell
http://www.novell.com/linux/

SuSE was taken over by Novell in 2003. The SuSE brand is still used for some distributions. Novell branded Linux distributions consist of the core SuSE distribution and some additional components or branding.

As with Red Hat, Novell provides support for it's Enterprise distributions for long periods (http://support.novell.com/lifecycle/index.jsp) - general support for SuSE Enterprise runs for 5 years.

SuSE has recently introduced OpenSuSE which is intended to be a similar project to Red Hat's Fedora being a volunteer-driven testing ground for future releases of SuSE / Novell Linux (http://en.opensuse.org/).

Others
There are a huge number of Linux distributions both commercial and non-commercial intended for a myriad of uses ranging from desktop to server to system recovery or security. http://lwn.net/Distributions/ lists 517 active distributions with a range of purposes, languages and target platforms.

Who is using Linux?

- Dot coms
  - Google
  - Amazon
  - Paypal
- Financial
  - Irish Stock Exchange
  - First Trust Corporation
  - Central Bank of India
- Entertainment
  - Ticketmaster
  - Pixar
  - Industrial Light and Magic

The Google Linux Cluster
http://www.researchchannel.org/program/displayevent.asp?rid=1680

Linux slashes costs for bank giant
http://www.computerweekly.com/articles/article.asp?liArticleID=135436

Pixar switches from Sun to Intel
http://news.com.com/2100-1001-983898.html

Red Hat Success Stories
http://www.redhat.com/solutions/info/casestudies/

SuSE Success Stories
http://www.novell.com/success/

Netcraft Web Server Survey
http://news.netcraft.com/archives/web_server_survey.html

Who's using Debian?
http://www.debian.org/users/

Getting Started

System Accounts

- Users
  - uid
  - username
  - password
- Groups
  - gid
- Files
  - /etc/passwd and /etc/shadow
  - /etc/group and /etc/gshadow
- Other authentication mechanisms
  - LDAP
  - Kerberos

A user account represents someone or something capable of using files on the system (can be either a person or a system process).

A group account is a list of user accounts. Each user account has a main group and as many others as is needed.

Users are defined in the /etc/passwd file. This file contains various information about users including their **login name**, **encrypted password**, **uid**, **gid**, **user's real name**, **home directory** and **shell**. Modern systems tend to store the users password in a separate file (/etc/shadow).

Groups are defined in /etc/group. This file contains the **group name**, **group password**, **gid** and a **list of users**.

Users and groups provide the operating system with a way of controlling access to system resources and maintaining an audit trail.

Linux systems can also use more advanced authentication systems such as **kerberos** or **LDAP authentication** (e.g. Active Directory) by adding **Pluggable Authentication Modules** (**PAM**) to the system.

Account Settings

- Passwords
  - choosing good passwords
  - password expiry
  - password security
- Shell
- Home directories

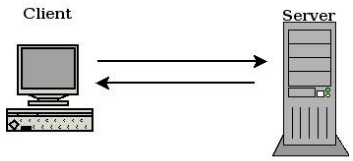
Each user account has a password. Different systems enforce different password policies but a good password is generally hard to guess and not a simple word (like Password!).

The **shell** or **command interpreter** is the program that takes your commands and does something with them. All user interaction with a Linux system is conducted through the shell (if using the console, users of a graphical environment can interact with the system without using the shell although one will still be associated with the user). There are a number of different shells available on a typical Linux system which use slightly different syntax.

Each user account is assigned their own directory within which to store their files. This directory is known as the **home directory**. Its location varies depending on the particular Linux version (**/home/*username*** and **/usr/users/*username*** are common locations).

Connecting

- Local
  - the console
- Remote
  - telnet
    - *telnet <hostname> [port]*
  - ssh
    - *ssh <username>@<hostname>*
  - rlogin, rsh
    - *rlogin -l <username> <hostname>*



In order to use a Linux system you must either login to it from the **console** or connect to it over the network. The console consists of one or more **virtual terminals** which can be accessed via a keyboard and monitor (you can switch between virtual terminals using ALT-F1..Fn).

Connections over the network are also know as **remote connections** (versus **local connections**). There are a number of standard **protocols** used to connect to systems. All of them allow you to connect over a **TCP/IP network** such as the Internet. Once connected, you can enter commands on the computer as if you were sitting at that computer. Normally, before you can execute any commands, you need to **login** with your **username** and **password**.

The **telnet** command uses a simple text protocol for connecting to remote systems. The telnet command is provided on most operating systems. Its big disadvantage is that it does not provide any encryption during a session so all data including usernames and passwords can be read by others on the network. The telnet command can also be very useful for diagnosing network problems.

The **ssh** command provides secure access to systems over a network. It uses cryptographic technology to **encrypt** any data you send (including usernames and passwords). You can also optionally use **public key authentication** rather than sending passwords. The ssh suite also provides the **scp** and **sftp** commands for secure copying of files.

The **rlogin** and **rsh** commands are another older plaintext technology which use the **Berkeley rhosts** file for authentication and authorisation. The rhosts file contains a list of hostnmes from which a connection to a server is allowed. If the rhosts file is unavailable, rlogin and rsh fall back to using passwords like telnet.

Copying files over the network

- ftp
  - operation
  - transfer mode
  - anonymous ftp
  - plaintext usernames and passwords
- sftp
  - same interface as ftp
  - encrypted usernames and passwords
- scp
- rcp

File Transfer Protocol (**FTP**) is one of the original methods used on unix systems to copy files from one system another over the network. It uses a client/server model. When you wish to transfer files, you (the client) connect to a server (the system you want to upload files to or download files from) using the following syntax,

```
ftp <server name>
```

The server responds with prompts for a username and a password. Successful authentication places the user at the ftp prompt,

```
ftp>
```

From here, a user can **ls**, **cd**, **get <file>** or **put <file>**. Multiple files can be sent or received using **mput <files>** or **mget <files>**. It is recommended to always use **bin**ary transfer mode (see file endings). When finished, signal to the server to **close** the connection.

The ftp command sends usernames and passwords in the clear over the network which is insecure. **anonymous ftp** is a form of passwordless ftp.

The **sftp** command is a secure version of ftp built on top of ssh. It uses virtually the same syntax.

```
scp <filename> <username>@<hostname>:/tmp
```

```
rcp <filename> <username>@<hostname>:/tmp
```

System documentation (1/2)

- Distribution-specific documentation
  - Administration and install guides
  - User manuals
  - Reference guides
  - Release Notes
- man pages and the man command
  - *man [section] <page>*
  - *man -k <topic>*
  - *man -f <topic>* or *apropos <topic>*
  - *man man!*

Linux and UNIX systems in general provide a lot of online documentation in electronic formats.

All distributions are supplied with high quality manuals and reference documentation.

http://www.redhat.com/docs/manuals/enterprise/ provides guides for installation on a number of platforms, a system administration guide, a general reference guide and a number of security documents and release notes for each release of Red Hat Enterprise Linux.

Novell provides documentation for SuSE at *http://www.novell.com/documentation/suse.html* and documentation for Novell Desktop Linux at *http://www.novell.com/documentation/nld/* including a quickstart guide, a deployment guide and guides for various components of the system including KDE and GNOME.

Debian provide a full set of documentation including an installation guide, user manual and reference at *http://www.debian.org/doc/*.

The original approach to providing information on UNIX systems was the **man** command. man provides access to the information on commands, system calls, special files, file formats and others. The man pages tend to contain lots of good information but are not always very user-friendly. The man pages are split up into sections

1 Executable programs or shell commands
2 System calls (functions provided by the kernel)
3 Library calls (functions within system libraries)
4 Special files (usually found in /dev)
5 File formats and conventions eg /etc/passwd
6 Games
7 Macro packages and conventions eg man(7), groff(7).
8 System administration commands (usually only for root)
9 Kernel routines [Non standard]

System documentation (2/2)

- GNU info
- <command> -h, --help
- The Linux Documentation Project
  - http://www.tldp.org/
  - FAQs, HOWTOs, Guides
- Project specific documentation for Samba, Apache and others.

The GNU project distribute most of their manuals and reference pages in the info format. The standard tool for reading info files is the **info** command which isn't very user-friendly. The info pages tend to contain a lot of good information so it is worth getting familiar with the info command (or trying one of the alternatives such as **tkinfo**).

Most commands have some basic help built-in which can be accessed by invoking the command with either `-h` or `-help`. If in doubt, try both.

The Linux Documentation Project is a volunteer driven project found on the web at *http://ww.tldp.org/*. The project has assembled a large collection of high quality **HOWTOs**, **Guides** and **FAQs**.

Finally, a number of the larger applications used on Linux including Samba, the Apache webserver and Apache Tomcat have their own websites dedicated to providing documentation and support for these applications.

Samba
http://www.samba.org/samba/docs/

Apache Webserver
http://httpd.apache.org/docs-project/

Apache Tomcat
http://jakarta.apache.org/tomcat/

Exercise 1.1 – Getting started

1) Start up *telnet* on your local machine.
2) Start up *putty* on your local machine.
3) Login to the server using
- **ssh**
- **telnet**

4) Upload a test file from your local system to the server using each of the following methods,
- **ftp**
- **sftp**
- **scp**

How to start telnet
- Click on Windows start button
- Select *Run Command ...*
- Type in *telnet*

Putty
- Putty is a free windows ssh client available to download from
  http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html

How to exit various commands
- telnet – type *CTRL-]* to enter command mode nd type *quit*
- man – press *q*
- info – *CTRL-x* + *CTRL-c*

How to view files
- when logged in, type *more* <*filename*> (*q* to exit).

Software for transferring files
- Filezilla (ftp and sftp) - http://filezilla.sourceforge.net/
- Winscp (scp and sftp) - http://winscp.net

The full Putty install kit also includes a command-line scp command called *pscp*.

Exercise 1.2 – Getting started

5) Use man to retrieve information on each of the following:

- **the `mail` command**
- **the `passwd` file (not the `passwd` command)**
- **the `printf` system call (not the `printf` command)**

6) Use info to review the *info primer*.

7) Find the following on TLDP:

- ***The Unix and Internet Fundamentals HOWTO***
- ***Advanced Bash-Scripting Guide***
- ***The Linux FAQ***

Atlantic LINUX

The Shell

What is the shell?

- Command-line interpreter
- Scripting environment
- Features
  - built-in commands
  - aliases
  - variables
  - pipes
  - input/output redirection
  - history
  - tab completion
- Shell commands and variables are CASE SENSITIVE

The purpose of any **shell** or **command-line interpreter** is to provide the user with a way of interacting with the operating system. Every shell allows a user to run operating system **commands**. Shells also usually provide a minimal set of **built-in commands** to perform basic tasks (such as navigating the filesystem and managing files). Shells also allow common command sequences to be **aliased** to shorter strings.

Shells use **variables** to store information temporarily and pass it between **processes**. A variable is a named storage location in system memory. Each shell also has a set of special variables known as **environment variables** which control the behaviour of the shell and allow the shell to return information about the system.

Shells can also be used to write **scripts** which are interpreted by the system when they are executed. **Shell scripts** are typically used to perform basic system management or text processing tasks.

There are different flavours of shell including the **bourne shell**, the **c shell** and the **korn shell**. They all possess the same basic characteristics but differ in the the details of their built-in commands, variables and syntax used in their scripts.

Most shells store a **history** of the most recently used commands which can be retrieved later on (often using the arrow keys). The history is usually written to a temporary file in the user's home directory.

Some shells include a feature called **tab-completion** – if you type part of a command or filename at the prompt, pressing the **tab** key will cause the shell to either complete the command or filename if there is only one possible completion or list the alternatives if there are multiple possible completions.

Choosing a shell

- Bourne shells
  - sh
  - bash
- C shells
  - csh
  - tcsh
- Korn Shell
- Others

sh, the **bourne shell** was the original UNIX system shell. It contains all the basic features of a shell and is commonly used in shell scripts due to its portability across Linux and many different UNIX systems.

bash, the **bourne again shell** is a successor to the bourne shell developed by the GNU project. It contains all the features found in the bourne shell and incorporates some aspects of both the **korn shell** and the **c shell**. Bash has become the de facto standard shell on Linux systems and is a reasonable alternative to sh for scripting if the script is only intended for Linux systems.

The **c shell**, **csh** uses a command syntax similar to the C Programming language. Like the bourne shell, it is found on practically all Linux and UNIX systems. It is not recommended for shell scripting due to problems file descriptors, flow control, white space, signals and quoting

Tcsh, is an enhanced, but completely compatible version of csh. Any script written in csh will behave exactly the same when executed with tcsh. Unfortunately this also means it has the same limitations as csh for shell scripting. Tcsh is a very user-friendly shell and contains a similar feature set to bash.

ksh, the **korn shell** is a bourne shell derivative with many enhancements. It is common on UNIX systems but a version is also available on most Linux systems (though not as common as either bash or tcsh). There are two major versions of the Korn shell in common use, the 1988 version (ksh88) and the 1993 version (ksh93). Many systems only provide the 1988 version because of licensing issues with the 1993 version. There is also a shell called the **Public Domain Korn Shell** (**pdksh**), which implements most ksh88 features and some ksh93 features – this is the typical Korn shell found on Linux systems.

Switching to a different shell

- Changing shells
  - `chsh`
  - `/etc/shells`
- Restricted shell
  - changing directory
  - changing shell
  - changing shell variables
- Changing user details
  - `chfn`
  - `.plan`

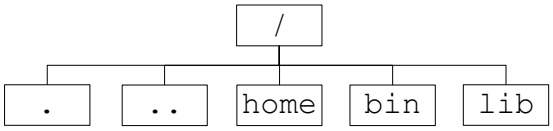
Any user can change their shell to any of the other shells available on the system using the **chsh** command. Typically, a user can only change to one of the permitted shells listed in **/etc/shells**.

Accounts may sometimes be configured to use a **restricted shell** which is one of a number of shells which provide less functionality than usual including not being able to change directories, change your shell, alter shell variables and so on.

The **chfn** program can also be used to change various properties of your system account including various contact details such as your name, phone number and office number.

Navigating the filesystem

- The filesystem tree



- Commands
  - `pwd`
  - `cd`
  - `ls`
- Inodes

The concept of a **filesystem tree** is at the core of Linux and UNIX systems. All files are organised in a tree structure with the **root directory** or / at the top of the tree. Directories are organised hierarchically underneath this. There are 2 special directories **.** and **..** which occur at every level of the tree. These are used as shorthand to specify the **current directory** (**.**) and the **parent directory -** the directory next closest to the root in the tree (**..**). The Linux filesystem tree contains a number of standard directories and sub-directories.

The **pwd** command prints the **current working directory** (**cwd**). Most shells include this as a built-in but the system also provides a pwd command. Some versions of the pwd command differentiate between the physical directory while others follow **symbolic links**.

The **cd** command changes the current working directory to the one specified or the one defined in the *$HOME* variable if no directory is specified. The cd command can use **relative** or **absolute** paths. If relative paths are specified, the behaviour of the cd command is controlled by the *$CDPATH* variable. *$CDPATH* defaults to the current directory.

The **ls** command lists the contents of the current working directory (files and directories). The command takes a wide range of switches which enhance the listing. **-l** (use a long listing format displaying permissions, owner, group, size, modification date and filename) and **-a** (display file entries beginning with .) are 2 useful switches.

At the filesystem level, each file is represented by a data-structure called an **inode** which various fields including file **mode**, **owner information**, **size**, **timestamps** and a series of **pointers** to the **blocks** containing the actual **file data**.

Shell Variables

- Shell variables
- Environment variables
- Viewing variables with `echo`
- Listing variables
  - `set`
  - `export`
  - `setenv`
  - `env`

An ordinary shell variable comes into existence when a value is assigned to it. The method of assigning a value to a variable differs slightly between the bourne shell family and the c-shell family.

Linux maintains a special collection of variables called the environment. When a new process is created, this environment is copied from its parent. A new process is created every time a command is run from the shell prompt. An extra step is usually needed to set an environment variable with the syntax also varying between shells.

Ordinary shell variables and environment variables can be viewed using the **echo** command as follows

```
echo $VARIABLE_NAME
```

To view all variables that have been set in the **bourne shell**, use the **set** command with no arguments. To view only environment variables, use the **export** command with no arguments.

To view all variables that been set in the **c shell**, use the **set** command with no arguments. To view only environment variables, use the **setenv** command.

The **env** command can also be used to view your environment variables.

Giving variables values

- In the bourne shell
  - `VAR=value`
  - `export VAR`
  - `export VAR=value`
- In the c-shell
  - `set VAR=value`
  - `setenv ENV value`
  - synchronised variables
- `unset`

With the **bourne shell** and derivatives (sh, bash, ksh) – a variable is set using the following syntax

```
VARIABLE_NAME=value
```

If you wish to make this variable available to the **environment** – the variable needs to be **exported** using the following syntax

```
export VARIABLE_NAME
```

The bash and korn shells allow these two steps to be combined as

```
export VARIABLE_NAME=value
```

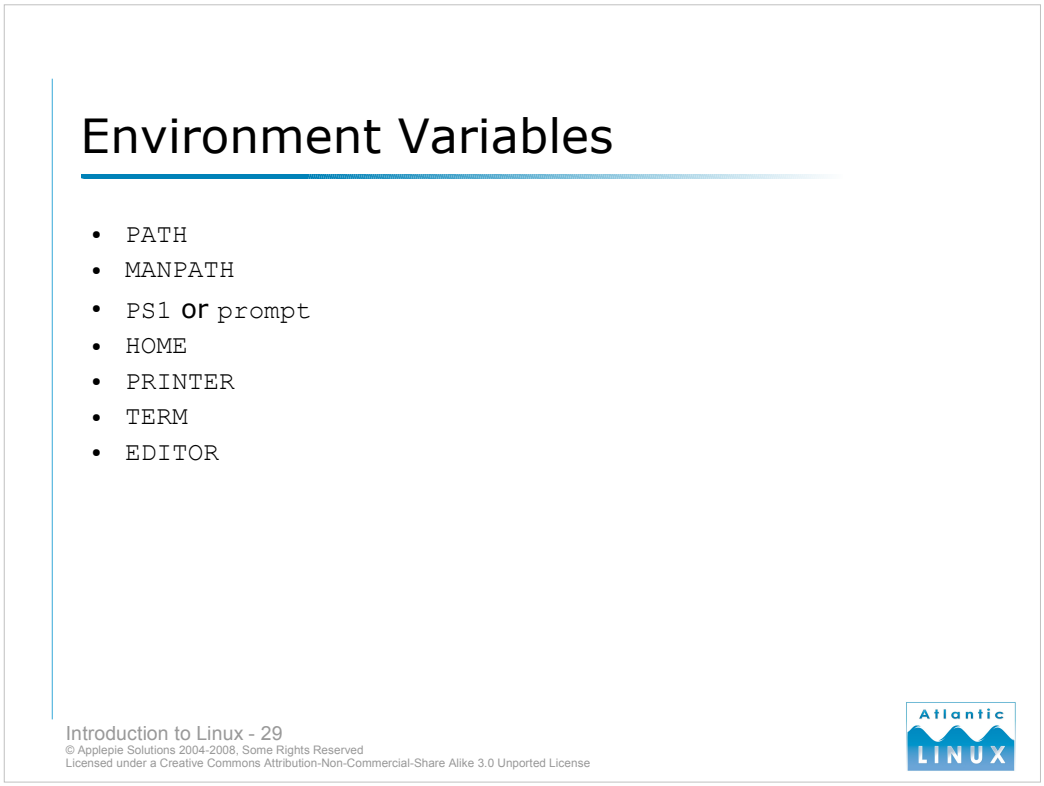
The **c-shell** uses a slightly different syntax for setting variables as follows

```
set VARIABLE_NAME = value
```

To make a c-shell variable available to the environment requires the use of a different command as follows

```
setenv ENVIRONMENT_VARIABLE_NAME value
```

tcsh synchronises some shell and environment variables when one or the other is changed including path and PATH, cwd and PWD, term and TERM and others. It is generally best to set the shell variable and let the shell set the environment variable.



There are many environment variables which both control various aspects of shell behaviour and return information about parts of the shell's configuration. The following are some of the more common useful ones. A full list of a shell's environment variables can be found in the man page for that shell.

PATH – This is a colon separated list of directories in which to search for commands. When a user types a command at the prompt, if it does not start with /, the shell will search each directory in the PATH for this command and execute it if it finds it.

e.g.

```
$ PATH=/usr/gnu/bin:/usr/local/bin:/bin:/usr/bin:.
$ export PATH

> set path (/bin /usr/bin .)
```

PS1 – this shell sets the format of the shell prompt (and includes various special parameters). In the c shell, the equivalent variable is **prompt**.

HOME – Initialised to the home directory of the user. This controls the default behaviour of **cd** and some properties of the prompt.

PRINTER – this sets the default printer for printing commands to use.

TERM – This sets the terminal type which controls the behaviour of various output programs.

EDITOR – this variables specifies the editor used by commands like **passwd**

Paths

- Absolute paths
  - `/bin/ls`
  - `/etc/passwd`
- Relative paths
  - `./ls`
  - `../ls`
  - `ls`
- Security problems with relative paths
- Use the `which` command to verify your PATH
- System administrators and scripts should always use absolute paths!

Files, including commands and directories, can be specified using either **absolute paths** or **relative paths**.

An absolute path is one that describes the file using the entire path to that file in the filesystem tree. Absolute paths can be recognised by the fact that they start with /
e.g. /bin/ls works from the / directory, to the bin subdirectory and then invokes the ls file in the bin subdirectory.

Relative paths describe the file using a path *relative* to the shell's current location in the filesystem (and may also available of the shell's knowledge of special variables such as PATH).
e.g ../../bin/ls moves up through 2 levels of the filesystem and then descends into the bin subdirectory (which may or may not be /bin depending on the shell's current working directory) from where it invokes the ls file.

A special case of relative paths is where a command is typed at the shell prompt without specifying any path. The shell typically searches the directories listed in the PATH environment variable for a file matching the specified command. The first file found in this search is invoked.

You can use the *which* command to test which command or executable will be invoked if you type a command at the shell prompt – if it finds a matching command in your PATH, it will display the full path to the command.

For security reasons, system administrators or administration programs and scripts should always use absolute paths for commands. There are a number of potential security problems with using relative paths or relying on the contents of PATH.

Some Useful Commands

- Printing output to the screen
  - `echo`
- Viewing files
  - `cat`
  - `more` or `less`
- Identifying files
  - `file`
- Finding commands
  - `which`
- Command-line options
  - short (`-h`)
  - long (`--help`)

The **cat** command is used to concatenate files and print them to the screen. With no arguments, cat displays **standard input** to **standard output** (which is of limited use). By passing one or more files as arguments to cat, the contents of these files are combined and sent to standard output. **Redirection** and **pipes** can be used to send output to destinations other than standard output. Note that it can be difficult to read files which span more than one screen with cat.

more and **less** perform the same function as cat but pause after displaying each screen of information. less is a more advanced version of more, providing enhancements such as the ability to move backwards and forwards in a file, faster startup times and better terminal support. You can use *page up, page down, space* and the arrow keys to move around a file viewed with less or more. To exit press *q* or *ESC*.

The **file** command determines the type of a file by performing a set of standard tests against the file contents. File can distinguish text files in various formats and character sets, executables in various formats and a myriad of data files.

Many commands used on Linux take **options** which modify the behaviour of the command or enable additional features. Options are usually specified after the command name. Most GNU commands take a short cryptic form of an option or a longer more user-friendly form of the same option e.g. **-h** or –**help**. Short options are usually identified by a single preceding dash and long options by a pair of preceding dashes.

Exercise 2.1 – The Shell

1) **Navigate to the following directories and list the contents of the directory,**
   - `/home`
   - `/usr/bin`
   - `/etc`
2) **Use absolute paths to list the contents of those directories without navigating to them first.**
3) **Display the contents of the following variables and explain what they mean:**
   - `SHELL`
   - `USER`
   - `PWD`

The command to list files in a directory is *ls* (more information on this in the next section) – the basic usage is *ls* (to list the contents of the current working directory) or *ls* <*directory*> to list the contents of the specified directory.

Exercise 2.2 – The Shell

4) Use a system command to determine what type of file or directory each of the following is:
   - `/`
   - `/etc/passwd`
   - `/bin/ls`
   - `/lib/libc-2.3.5.so`
5) View the contents of `/etc/passwd` with `cat`, `more` and `less`.
6) Change your `PATH` variable to "/" and try running `cat`, `more` and `less` again. Explain what is happening.
7) Change your `PS1` (`bash`) or `prompt` (tcsh) variable.

Advanced Shell Topics

I/O Redirection

- Standard input (file descriptor 0, stdin)
- Standard output (file descriptor 1, stdout)
- Standard error (file descriptor 2, stderr)
- Redirection operators
  - `[n]> file`
  - `<[n] file`
  - `[n]>> file`
- Redirections are processed in the order they appear, from left to right.

By default, the shell takes input from the keyboard and sends output to the screen. The keyboard is called the **standard input** (**stdin, 0**) device. The screen is called the **standard output** (**stdout, 1**) device. There is also another output device called **standard error** (**stderr, 2**) which is also normally sent to the screen.

All commands default to taking input from standard input, sending normal output to standard output and sending error messages to standard error.

I/O redirection involves changing the default behaviour and taking output from a file, command, program or script and sending it as input to another file, command, program or script. The following **redirection operators** can be used to change a command's behaviour.

- > and >> are used to redirect stdout to a file. With either operator, if the file doesn't exist it will be created. With >, if the file exists, it is overwritten. With >>, if the file exists it will be appended to.
- 2> is used to redirect stderr to somewhere else.
- < is used to redirect stdin from a file.

Examples:
Write the output of the date command to a file called date.txt
```
$ date > date.txt
```

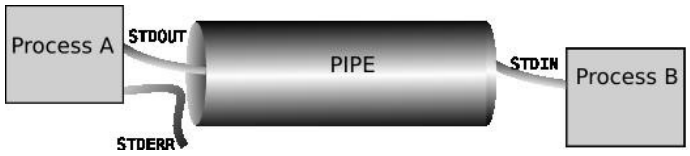
Append the output of the cal command to a file called date.txt
```
$ cal >> date.txt
```

Use the file data.txt as the input to the sort command rather than stdin.
```
$ sort < data.txt
```

Use data.txt as input and send the output to the file sorted-data.txt rather than stdout.
```
$ sort < data.txt > sorted-data.txt
```

Pipes

- |
- chaining commands
- `xargs`
- named pipes (fifos)



Pipes are also used for I/O redirection.

They behave in a similar fashion to > but are specifically used to redirect the output of a command to the input of another command.

This redirection allows multiple commands to be **chained together**, each one performing an operation on the output of the previous command e.g.

```
cat file.txt | tr a-z A-Z | sort
```

the *cat* command outputs the file *file.txt* which is input to the *tr* command which changes the case of each line from lowercase to uppercase. This uppercase output is used as input to the `sort` command which sorts the lines and outputs the result to stdout. The result could just as easily have been redirected to a file.

```
ps aux | grep java | grep -v grep
```

the *ps* command lists all processes running on the system. This list is sent to the *grep* command which lists lines matching the specified pattern (*java* in this case). The results of this command, are in turn sent to another command which lists only lines that do not match the pattern (-v option to grep).

Sometimes, when using pipes, it is possible to overflow the command-line buffer of a command by passing it too much information. So, with a chain like

```
commandA | commandB
```

If commandA generates a lot of output, it can overload commandB's input buffer (usually resulting in an error message like *Too many arguments*). The **xargs** is intended to circumvent this problem by only sending as much output to commandB as it can handle and invoking it multiple times if necessary to process all of the output from commandA e.g.

```
commandA | xargs commandB
```

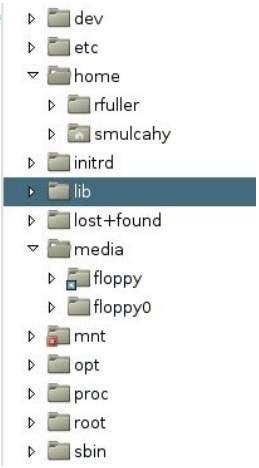
Exercise 3 – Advanced Shell

1) Write a command to list the contents of /etc/passwd to a file in /var/tmp (such as *mytest*).
2) Write a command to list the contents of /etc/group to the same file used in 1, without overwriting the original content.
3) Write a command to list the contents of /etc/passwd to a file in /var/tmp while redirecting any errors to a different file in /var/tmp.
4) Explain what the following will do (you may need to use the *man* command to check what a particular command does):
   - `cat /etc/passwd | tr a-z A-Z`
   - `ls /etc | sort | tr A-Z a-z`

Files

Linux Files

- Everything is a file!
  - Docs, pictures, executables
  - Directories
  - Devices
  - Kernel internals
  - .dot files
  - hard links
  - soft (symbolic) links
- Filesystem Hierarchy Standard (FHS)
- Hard and soft links



Atlantic LINUX

On a Linux system, everything is a file. This includes the usual files such as documents, images, and commands. It also includes *devices*, *kernel internals* and *system settings*. Representing everything as a file is a programming abstraction which allows users to manipulate various parts of the system using a standard interface.

The filesystem tree is organised in a standard way known as the **Filesystem Hierarchy Standard** (**FHS**) which allows users of any Linux distribution (or developers of applications for Linux systems) to always find the same kinds of files in the same location.

A file's **inode** (the data structure on the disk that represents the file) can have more than one filename associated with it (a filename is just an entry in a directory data structure). The inode for the file keeps a count of how many filenames are associated with it (the link count). You can add additional filenames pointing to an inode by **hard linking** the new filename against the original filename. This increases the link count of the file. The **ln** command is used to link, passing the *<original filename>* and the *<new filename>* as arguments. A file is not deleted from the filesystem until its link count is 0.

e.g.

```
ln original_file.txt new_file.txt
```

You can also create **soft links** or **symbolic links** to files. These are more like **shortcuts** in the Windows operating system, they are aliases to the original files. You can create with them with **ln -s**. The original file can still be deleted when it has one or more soft links pointing to it.

e.g.

```
ln -s original_file.txt softlink_to_file.txt
```

File Hierarchy Standard - /

- /
- /bin
- /boot
- /dev
- /etc
- /home
- /lib

The current version of the FHS is 2.3 (29-Jan-2004). It is an evolving standard developed by the Linux community to address the needs of users, programmers, system administrators and software vendors. It describes the key features of the filesystem including where specific types of files should be found and the purpose of various system directories and sub-directories. The FHS is a component of a larger set of standards – the Linux Standard Base (LSB). The following contains some excerpts from FHS 2.3

/ - *The contents of the root filesystem must be adequate to boot, restore, recover, and/or repair the system.*

/bin - *contains commands that are required when no other filesystem is available. This can include commands used by system administrators and users.*

/boot - *contains files used in the boot process.*

/dev – *contains special device files.*

/etc - *contains configuration files. A configuration file is a local file used to control the operation of a command or system program.*

/home – *An optional location for users home directories. Alternatives like **/usr/users** are also permitted by the standard.*

/lib - *contains those shared libraries used to boot the system and run the commands in the root filesystem (the commands in /bin and /sbin).*

File Hierarchy Standard - /

- /mnt (and the mount command)
- /opt
- /root
- /sbin
- /tmp
- /usr
- /var

Atlantic LINUX

/mnt – *this directory is used as a temporary location for mounting other filesystems*.

/opt – *intended to be used for the installation of add-on application software packages. Applications are typically installed in a subdirectory named after the application.*

/root – *this is the recommended default location for user root's home directory*.

/sbin – *the location of commands used for system administration. This directory contains commands and other files essential for booting, restoring, recovering and/or repairing the system.*

/tmp – *this directory is used by programs which require somewhere to create (typically small) temporary files.*

/usr – *the /usr hierarchy of directories is intended as a store of* ***shareable***, ***read-only data***. *Any information which is specific to a particular system or changes over time is stored elsewhere.*

/var – *the /var hierarchy of directories is intended as a store of* ***writable*** *data. Files that are written to during normal system operation are usually stored under here rather than under /usr. Some files under /var are* ***shareable***, *some* ***non-shareable.***

File Hierarchy Standard - /usr

- /usr/bin
- /usr/include
- /usr/lib
- /usr/local
- /usr/sbin
- /usr/share
- /usr/src

Atlantic LINUX

/usr/bin – *the main location for user commands.*

/usr/include – *the location of the system include files for the C programming language.*

/usr/lib – *the main location for system libraries, object files and some system commands that aren't called directly by users or scripts.*

/usr/local – *a hierarchy used for software and data installed only on this system or used by a small group of systems. Distribution software is never installed into this hierarchy so it is a safe location in which to install add-ons such as the JDK (safe in the sense that subsequent upgrades to the distribution will not overwrite or erase software installed under /usr/local).*

/usr/sbin – *this directory contains any system commands that aren't needed for system booting, repair, recovery or restore. Only commands used exclusively by the system administrator should be installed here.*

/usr/share – *this directory is intended to store read-only, architecture independent data files. This directory could be shared between multiple systems running the same OS version. /usr/share is not intended to be shared between different versions of the same OS or different OSs.*

/usr/src – *this is an optional directory in which to place read-only copies of source code. The Linux kernel source code is typically installed here by Linux distributions.*

File Hierarchy Standard - /var

- /var/cache
- /var/lib
- /var/lock
- /var/log
- /var/mail
- /var/opt
- /var/run
- /var/spool
- /var/tmp

Atlantic LINUX

/var/cache - *intended for use by applications to store cached data. This should only be data which the application can safely regenerate if deleted.*

/var/lib – *subdirectories under /var/lib are intended to store variable state information used by applications and the system itself. This data stored here is typically critical to the operation of a program but changes over time.*

/var/lock – *used to store system and application-specific lock files.*

/var/log – *used to store system and application log-files. The main system logs are normally stored in /var/log while other system applications store their logs in appropriate subdirectories.*

/var/mail – *this contains the system mail spool or user mailboxes where user mail is initially delivered.*

/var/opt – *this is used by packages installed in /opt when generating variable data.*

/var/run – *used to store run-time system and application data. This directory and any subdirectories usually have their contents erased at system boot.*

/var/spool – *this is a staging area for application data which the application has yet to be process. Data in this directory is usually managed by the application and will not be normally be deleted at boot time.*

/var/tmp – *this directory is intended for storing temporary files which are deleted at boot-time (as opposed to /tmp). /var/tmp should always be used for large temporary files.*

ls command

```
ls -la /bin
```

```
total 3404
drwxr-xr-x   2 root root   4096 Oct 13 18:46 .
drwxr-xr-x  25 root root   4096 Sep  3 20:05 ..
-rwxr-xr-x   1 root root   2684 Dec 24  2002 arch
-rwxr-xr-x   1 root root  82312 Apr  3  2002 ash
-rwxr-xr-x   1 root root 511400 Apr  8  2002 bash
-rwxr-xr-x   1 root root  16504 Jul 16 12:37 cat
-rwxr-xr-x   1 root root  31404 Jul 16 12:37 chgrp
-rwxr-xr-x   1 root root  31212 Jul 16 12:37 chmod
-rwxr-xr-x   1 root root  34572 Jul 16 12:37 chown
-rwxr-xr-x   1 root root  51212 Jul 16 12:37 cp
-rwxr-xr-x   1 root root  49092 Nov 24  2001 cpio
```


The **ls** command has many options which control its output. The most common options used with it are **-l** and **-a** which tell the ls command to display **more information about each file** and **display entries starting with . (dot)** respectively.

Files starting with . are described as **dot-file** and are usually used to store per-user system or application settings. Most applications create a dot-file to store their settings.

The *total 3034* refers to the total space used in this directory in kilobytes – this does not include space used by the contents of any subdirectories.

The **-i** option can be used to view the **inode numbers** for each file.

ls command

permissions

```
total 3404
drwxr-xr-x   2 root root   4096 Oct 13 18:46 .
drwxr-xr-x  25 root root   4096 Sep  3 20:05 ..
-rwxr-xr-x   1 root root   2684 Dec 24  2002 arch
-rwxr-xr-x   1 root root  82312 Apr  3  2002 ash
-rwxr-xr-x   1 root root 511400 Apr  8  2002 bash
-rwxr-xr-x   1 root root  16504 Jul 16 12:37 cat
-rwxr-xr-x   1 root root  31404 Jul 16 12:37 chgrp
-rwxr-xr-x   1 root root  31212 Jul 16 12:37 chmod
-rwxr-xr-x   1 root root  34572 Jul 16 12:37 chown
-rwxr-xr-x   1 root root  51212 Jul 16 12:37 cp
-rwxr-xr-x   1 root root  49092 Nov 24  2001 cpio
```


The **permissions** column contains information about what **users** have access to a particular file and what **access** each type of users have.

1	2	3	4	5	6	7	8	9	10
File Type	User Permissions			Group Permissions			Other Permissions		
	read	write	execute	read	write	execute	read	write	execute
d	r	w	x	r	-	x	r	-	x

The first letter (d in this case) describes the type of the file. Types include a normal file (**-**), a directory (**d**), a **block device** (**b**), a **character device** (**c**), a **symbolic link** (**l**), a **socket** (**s**) or a **named pipe** (**p**)..

After this, the permissions on the file are specified in 3 groups of 3. The first group describes the permissions that the owner of the file has. The second group describes the permissions members of the file's group have. The third group describes the permissions that others have.

For each group, the first letter describes the **read permission** (**r** if read permission is granted, **-** otherwise), the second letter describes the **write permission** (**w** if write permission is granted, **-** otherwise) and **execute permission** (**x** if execute permission has been granted, **-** otherwise). Note that in the case of directories, execute permission signifies that the directory can be accessed by that party, while for commands, the execute permission indicates that the command can be executed.

ls command

owner and group

```
total 3404
drwxr-xr-x   2 root root   4096 Oct 13 18:46 .
drwxr-xr-x  25 root root   4096 Sep  3 20:05 ..
-rwxr-xr-x   1 root root   2684 Dec 24  2002 arch
-rwxr-xr-x   1 root root  82312 Apr  3  2002 ash
-rwxr-xr-x   1 root root 511400 Apr  8  2002 bash
-rwxr-xr-x   1 root root  16504 Jul 16 12:37 cat
-rwxr-xr-x   1 root root  31404 Jul 16 12:37 chgrp
-rwxr-xr-x   1 root root  31212 Jul 16 12:37 chmod
-rwxr-xr-x   1 root root  34572 Jul 16 12:37 chown
-rwxr-xr-x   1 root root  51212 Jul 16 12:37 cp
-rwxr-xr-x   1 root root  49092 Nov 24  2001 cpio
```


There can be only one **owner** of a file. The owner can be any of the users of a system or one of the special system accounts.

The **group** can be used to provide limited access to files.

Note also that the column before the owner and group contains the **link count** for each file (ref: hard links).

ls command

size in bytes

```
total 3404
drwxr-xr-x   2 root root    4096 Oct 13 18:46 .
drwxr-xr-x  25 root root    4096 Sep  3 20:05 ..
-rwxr-xr-x   1 root root    2684 Dec 24  2002 arch
-rwxr-xr-x   1 root root   82312 Apr  3  2002 ash
-rwxr-xr-x   1 root root  511400 Apr  8  2002 bash
-rwxr-xr-x   1 root root   16504 Jul 16 12:37 cat
-rwxr-xr-x   1 root root   31404 Jul 16 12:37 chgrp
-rwxr-xr-x   1 root root   31212 Jul 16 12:37 chmod
-rwxr-xr-x   1 root root   34572 Jul 16 12:37 chown
-rwxr-xr-x   1 root root   51212 Jul 16 12:37 cp
-rwxr-xr-x   1 root root   49092 Nov 24  2001 cpio
```


ls on Linux systems defaults to listing the size of each file in bytes. Use of the **-h** option changes this to more user-friendly output (which varies between kilobytes, megabytes, gigabytes and terabytes depending on the actually size of the files).

ls command

modification date (normally!)

```
total 3404
drwxr-xr-x   2 root root   4096 Oct 13 18:46 .
drwxr-xr-x  25 root root   4096 Sep  3 20:05 ..
-rwxr-xr-x   1 root root   2684 Dec 24  2002 arch
-rwxr-xr-x   1 root root  82312 Apr  3  2002 ash
-rwxr-xr-x   1 root root 511400 Apr  8  2002 bash
-rwxr-xr-x   1 root root  16504 Jul 16 12:37 cat
-rwxr-xr-x   1 root root  31404 Jul 16 12:37 chgrp
-rwxr-xr-x   1 root root  31212 Jul 16 12:37 chmod
-rwxr-xr-x   1 root root  34572 Jul 16 12:37 chown
-rwxr-xr-x   1 root root  51212 Jul 16 12:37 cp
-rwxr-xr-x   1 root root  49092 Nov 24  2001 cpio
```


The date displayed in the ls -l output is generally the date on which the file was **last modified** but various options to ls can change this to other values (including the **file creation date**). Note that the format changes with the age of the file.

ls command

file name

```
total 3404
drwxr-xr-x   2 root root   4096 Oct 13 18:46 .
drwxr-xr-x  25 root root   4096 Sep  3 20:05 ..
-rwxr-xr-x   1 root root   2684 Dec 24  2002 arch
-rwxr-xr-x   1 root root  82312 Apr  3  2002 ash
-rwxr-xr-x   1 root root 511400 Apr  8  2002 bash
-rwxr-xr-x   1 root root  16504 Jul 16 12:37 cat
-rwxr-xr-x   1 root root  31404 Jul 16 12:37 chgrp
-rwxr-xr-x   1 root root  31212 Jul 16 12:37 chmod
-rwxr-xr-x   1 root root  34572 Jul 16 12:37 chown
-rwxr-xr-x   1 root root  51212 Jul 16 12:37 cp
-rwxr-xr-x   1 root root  49092 Nov 24  2001 cpio
```


The last column in the ls -l output is the actual name of the file.

File permissions and ownership (1/2)

- Setting permissions
  - Symbolic mode

    chmod <groups> <add/remove/set> <permissions> <file>

    e.g.

    chmod u=rwx,g=rx,o=rx foo

    chmod u=r,g=r,o= foo

  - Octal mode

    chmod <mode> <file>

    e.g.

    chmod 0755 foo

    chmod 0440 foo

The **chmod** command is used to change file permissions. It can be used in two different ways - **symbolic mode** or **octal mode**.

Using chmod in symbolic mode, it is invoked with the following format

```
chmod <groups> <add/remove/set> <permissions> <file>
```

groups is one or more of **u**,**g**,**o** and **a** where

- **u** is the user that owns the file
- **g** is other users in the file's group
- **o** is other users not in the file's group
- **a** is all users.

Permissions are added using the + operator, removed using the – operator or set (over-riding any existing permissions) using the = operator.

Permissions consists of one or more of the letters **rwxXstugo** for **read** (**r**), **write** (**w**), **execute** (**x**), **execute only if the file is a directory or already has execute permission for some user** (**X**), **set user or group ID on execution** (**s**), **sticky** (**t**), **the permissions granted to the owner of the file** (**u**), **the permissions granted to the group of the file** (**g**) or **the permissions granted to others** (**o**).

octal mode is a more powerful but less intuitive way of using chmod. Using chmod in octal mode, it is invoked with the following format

```
chmod <octal mode> <file>
```

octal mode is a series of 1-4 octal digits with values of 4, 2, 1 or 0. Missing digits are treated as 0. Multiple permissions are set at the same time by adding the values together. The first digit selects the **set user ID** (4), **set group ID** (2) and **stick** (1). The second digit selects permissions for the owner – **read** (4), **write** (2) and **execute** (1). The third digit selects permissions for the group (with the same values as the owner digit) and the fourth digit selects permissions for others.

File permissions and ownership (2/2)

- Changing file ownership
  - chown
- Changing file group
  - chgrp
- Changing file attributes recursively with `-R`

- Special permissions
  - The sticky bit
  - The setuid/setgid bit
  - File owner

File ownership is managed by 2 commands, **chown** for changing the **file owner** and **chgrp** for changing the file group.

chown and **chgrp** have a similar syntax. The are called as

<command name> <owner or group> <file>

for example,

```
chown smulcahy foo

chgrp users foo
```

You can invoke any of `chmod`, `chown` or `chgrp` against a directory and have settings changed recursively on all files and subdirectories within the directory by using the `-R` flag. Use this with caution, it can be difficult to undo the results of such an operation.

In addition to the normal read, write and execute permissions you can apply to files, there are 2 special permission bits which can be applied to files.

The **sticky bit** normally only has an affect when applied to directories. When it is set on a directory, any files in that directory can only be renamed or deleted by their owner (or the super-user). Without the sticky bit, any user with write permissions on a directory can rename or delete files in that directory. The sticky bit is useful for world-writable directories such as **/tmp** and **/var/tmp**.

There is a facility in Linux systems where a running command can make a system call to change either the user it is running as (**setuid()**) or the group it is running as (**setgid()**). Only the super-user can make these system calls. The operating system also allows files with the **setuid** or **setgid** bits set to run as their owner/group regardless of who invokes them.

File & directory commands

- File creation and timestamp updates
  - `touch`
- Copying and moving files
  - `cp`
  - `mv`
- Removing files and directories
  - `rm`
- Directories
  - `mkdir`
  - `rmdir`

The **touch** command **updates an existing files access and modification timestamps** to the current time. If the specified file does not exist, a new empty file is created.

The **cp** command is used to **copy** files. When invoked as *cp SOURCE DEST*, the SOURCE file is copied a new file called DEST. If DEST is a directory, SOURCE can be a list of one or more files which will be copied to the DEST directory.

cp -R can be used to copy entire directories and their contents.

The **mv** command is used to **move** or **rename** files. It uses the same syntax as the cp command, that is, *mv SOURCE DEST* renames the file called SOURCE to a file called DEST. Similarly, if DEST is a directory, SOURCE can be a list of one or more files (or directories) which will be moved to the DEST directory.

The **rm** command **removes** or **deletes** each file passed to the command as an argument. The rm command takes a number of options which should be used with care (**NB rm -rf**).

The **mkdir** command is used to **create directories** specified as arguments.

The **rmdir** command is used to **remove empty directories** specified as arguments.

Filename substitution

- Metacharacters
  - *
  - ?
- Character ranges
  - [...]
  - -
  - ^
- Home directory shortcut
  - ~

There are a number of **symbols** used by all of the common shells on Linux which have a special meaning to the shell. These special symbols are known as **wildcards** or **metacharacters** and provide a simple form of **pattern matching** (sometimes also called **globbing** or **filename substitution**). They provide a short-cut when you wish to apply a command to many files.

The most common wildcards which are supported by both bourne shell derivatives and c-shell derivatives are,

***** - the asterisk matches **any string including the null string**
? - the question mark matches **any single character**

[...] - the square brackets surrounding one or more characters **matches any one of the enclosed characters**
When using [], the "-" symbol is used to specify a range of characters e.g. [a-c] will match a, b and c. The "^" symbol is used to negate the range e.g. [^a-c] will match everything but a, b and c.

~ - the tilde symbol matches to the user's home directory

Filename substitution (examples)

1. `cd ~`
2. `ls /usr/bin/b*`
3. `ls /usr/bin/*zip*`
4. `ls /usr/bin/b[abc]`
5. `ls /usr/bin/b[a-c]`
6. `ls /usr/bin/?grep`

Atlantic LINUX

1. Change directory to the user's home directory.
2. List all files with b as their fire letter (including a file called “b”).
3. List all files containing the string “zip” (including a file called “zip”).
4. List any files called “ba”, “bb” or “bc”.
1. List any files called “ba”, “bb” or “bc” (same as 4 but using the range symbol).
2. List all files starting with any digit followed by the string grep (will match “egrep”, “fgrep” and “zgrep” but not “grep”).

Monitoring free space and inodes

- Files and directories
  - du
- Mounted filesystems
  - df
- -h option improves readability

The **du** command summarises the space usage of each file specified as an argument to it. It can used to check the space usage of both files and directories (it defaults to analysing directories). Usage is reported in kilobytes by default, the **-h** option makes the output for larger files more readable.

The **df** command reports filesystem disk usage. By default, df lists the space usage on all mounted filesystems. The **-h** is also supported by df.

Exercise 4.1 – Files

1) Create a file in your home directory containing a listing of all files in /bin that have filenames starting with "b".
2) Create a file in your home directory containing a listing of all files in /bin that have filenames not starting with "b".
3) Create a number of test files in your home directory with the following permissions (use a naming convention such as exercise3a, exercise 3b and so on):
   a) readable and writeable by you only
   b) readable and executable by you only
   c) readable, writeable and executable by you and readable and writeable by everyone else.

Atlantic LINUX

Exercise 4.2 – Files

4) Explain the permissions on the following files:
   - /bin/ls
   - /boot
   - /var/mail
5) Report the space used by your home directory and the free space available on your filesystems.
6) Where should large temporary files be placed according to the FHS?
7) Where would you expect a 3rd party application following the FHS to install its files to?

Processes

Processes and Threads

- Heavyweight process
- Lightweight process
- Speed of context switch
- Speed of creation
- Ease of sharing data
- Security
- NPTL

Multitasking operating systems use the concept of a **process**. A process is simply an instance of a running program. More modern systems have introduced the concept of **heavyweight processes** and **lightweight processes**. Traditional processes are the same as heavyweight processes. Lightweight processes are also known as threads.

A process can be said to consist of **code, data** and at least one **thread of execution**. Early UNIX process models contained only one thread of execution. Each process used its own **virtual address space** securely separating its code and data from all other programs executing on the system.

This separation comes at the price of a relatively large amount of time required to create new processes (since the memory needed by the process needs to be organised). In addition, when a system **context switches** between executing programs, it can take a relatively large amount of time to unload the data and code of a process and load the new process's data and code. Running multiple threads in the same context eliminates some of these problems.

Linux combines these two approaches – its **fork()** system call uses **copy on write** semantics to minimise the cost of process creation (a child process is initially given a pointer to its parent's memory which reduces the startup and context switching costs).

In addition to the lightweight process model, Linux provides a number of other threading libraries including a **Native POSIX Thread Library (NPTL)** for Linux. This attempts to address some of the shortcomings found in trying to map traditional UNIX thread models to the Linux model with early versions of Java for example.

Shell job control

- foreground jobs
- background jobs (&)
- listing jobs
- switching
- suspending
- interrupting

Most shells include at keast some basic **job control** which allows a user to create and manage multiple processes from a single prompt.

When a command is executed at the prompt, it is said to run in the **foreground**. The shell suspends and won't process any more input until a foreground job finishes (a job can be **interrupted** using CTRL-C). A job can also be **suspended** using CTRL-Z.

A command can also be run in the **background**. The command runs as normal but the shell will continue to accept additional input and can run other commands while the background jobs complete. A job can be run in the background by putting the & symbol at the end of the command. The shell normally displays the **job number** and **process number** when a background job is started.

The **jobs** shell command lists out all running and suspended background jobs.

A particular job can be brought to the foreground using the **fg** shell command and the job number preceded by the % symbol. A suspended background job can be resumed using the **bg** shell command.

Listing processes

```
# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
 root         1  0.0  0.0   1492   104 ?        S    Aug28   0:04 init [2]
 root         2  0.0  0.0      0     0 ?        SW   Aug28   0:00 [keventd]
 root         3  0.0  0.0      0     0 ?        SWN  Aug28   0:00 [ksoftirqd_CPU0]
 root         4  0.0  0.0      0     0 ?        SW   Aug28   6:05 [kswapd]
 root         5  0.0  0.0      0     0 ?        SW   Aug28   0:24 [bdflush]
 root         6  0.0  0.0      0     0 ?        SW   Aug28   0:10 [kupdated]
 root       222  0.0  0.0      0     0 ?        SW   Aug28   0:10 [kjournald]
 root       223  0.0  0.0      0     0 ?        SW   Aug28   0:00 [kjournald]
 root       294  0.0  0.0      0     0 ?        SW   Aug28   0:00 [khubd]
 daemon    1508  0.0  0.0   1604    64 ?        S    Aug28   0:00 /sbin/portmap
 root      1564  0.0  0.2   1628   360 ?        S    Aug28   0:56 /sbin/syslogd
 root      1603  0.0  0.0   2152    80 ?        S    Aug28   0:02 /sbin/klogd
 root      1607  0.0  1.8  12644  2324 ?        S    Aug28   0:00 /usr/sbin/named
```


The **ps** command provides a more complete view of the processes on a Linux system. Using ps, you can view all processes on the system rather than just those commands that you have executed from the prompt.

The ps command takes a large number of options to customise its output and include various fields. **ps ux** gives a reasonably informative listing of a users' processes including various details. **ps aux** uses the same output format to display all processes on the system.

Two other options to note are **f** for **forest** and **w** for **wide**.

ps aux columns

- **USER** is the **user name** of the process owner.
- **PID** is the **process ID**, a number that uniquely identifies a process (for the lifetime of that process).
- **%CPU** The average processor usage of the process.
- **%MEM** The percentage of physical memory used by the process.
- **VSZ** The amount of virtual memory allocated to the process in kilobytes (This will typically be much larger than the actual memory usage).
- **RSS** The resident set size – the amount of memory actually in use by the process.
- **TTY** The terminal (if any) that the process is attached.
- **STAT** The current **process state** (see next slide).
- **START** The start time or date of the process. This is the 24 hour clock time for the first 24 hours and the start date after that.
- **TIME** The amount of time this process has been executing on the CPU for.
- **COMMAND** The command that the process is executing ([thread]).

Process listing variations

- `ps -elf`
- `ps`
- `ps u`
- `ps ux`
- `ps x -o user,pid,ppid,cmd`

Atlantic LINUX

Process states

- Runnable (R)
- Sleeping (S)
- Uninterruptible Sleep (D)
- Traced or stopped (T)
- Defunct or zombie (Z)
- Additional BSD status codes
  - No resident pages (W)
  - High priority process (<)
  - Low priority process (N)
  - Process with pages locked in mem (L)

A process can be in various states on a Linux system. Typically the **kernel scheduler** gives each process a short period of execution time and then gives the next process a chance. This period of execution time is known as a **quantum** or **timeslice**. On Linux, it is normally about 100ms (but this varies widely depending on the kernel version, the system hardware and the scheduler in use).

When a process is ready to to be executed, it is added to the **run queue** and its state is set to **runnable** (**R**). If a process requests some resource which isn't currently available, such as access to an I/O device, it is put to **sleep** (**S)** until this resource becomes available.

Processes in **uninterruptible sleep** (**D**) have marked themselves as uninterruptible while performing some critical task (typically used by device drivers)or accessing a resource that must not be left in an unknown state. When the process is finished with this resource, it removes the uninterruptible flag. This flag is typically used by kernel related threads manipulating internal buffers. A process which appears in a **D** state in a process listing is usually suffering from an I/O problem of some sort – processes should only become uninterruptible for very brief periods of time while accessing I/O etc.

Stopped processes or processes being traced will be flagged as (**T**).

Zombie (**Z**) or **defunct processes** occur when the parent process of a child process has not called the wait() system call. A zombie process is actually only an entry in the process table which won't be cleared until wait() is called – they don't use any memory or cpu. A large number of zombie processes may indicate a problem on the system (parent processes dying for some reason).

Monitoring processes

- top cpu processes
- cpu state information
- similar details to ps
- process priority and nice

The **top** command is another useful alternative for monitoring processes running on the system. It provides a continuously updating list of the top processes in terms of cpu usage.

Top displays a section of statistics relating to CPU and memory usage and a list of the active processes on the system. It displays similar information to ps.

The **PID** field as the same as ps, providing the process id for that particular process. The **USER**, again, is the owner of the process. **PR** is the priority of a task. Tasks with a numerically lower priority value are given preferential treatment by the task scheduler.

The **nice** command can be used to change the priority of a process. The **NI** column reflects any changes made to a process using this approach.

VIRT is the size of a processes **virtual image** in kilobytes. It includes a processes code, data, shared libraries and any memory pages which have been **swapped out**. The **RES** column lists the **resident size** of a process in kilobytes. This is the physical memory currently in use by a task. The **SWAP** column lists the value of memory pages which have been swapped out in kilobytes.

$$VIRT = SWAP + RES$$

S is the process state as per ps.

%CPU and **%MEM** show the percentage of processor usage and physical memory usage of a process.

Signals

- What are they?
- What do they do?
- Sending signals via the keyboard
- Sending signals with the kill command
- Sending signals with system calls
- Common signals

The operating system uses a mechanism called **signaling** to send short messages to processes when it wishes to notify a process of an important event. Signals interrupt the normal execution of a process and force it to **handle** the signal immediately on receiving it. Typical events which might require a signal to be sent to a process include **floating point exceptions**, **termination signal from the user**, **suspend signals from the user**, **I/O errors** and so on. Signals are defined as integer values. A full list of signals is available in *signal(7)*.

Processes handle signals by passing execution to a **signal handler** routine which performs some specific activity in response to the signal. Typical actions a signal handler might perform include killing child processes and removing temporary files before terminating its own process. Some signals cannot be caught (SIGKILL, SIGSTOP).

When you use CTRL-C to interrupt a job or CTRL-Z to suspend a job, you are actually sending that process a signal (**SIGINT** in the case of CTRL-C and **SIGTSTP** in the case of CTRL-Z).

Arbitrary signals can be sent to any process using the **kill** command. The normal syntax is *kill <signal> <PID>*. Running kill with the **-l** option lists the available signals. A common use of kill is to send a SIGINT to a runaway process,

```
kill -9 1234
```

You can resume a suspended process using either the **fg** or **bg** commands. Both commands send a **SIGCONT** to the process.

Signals can also be sent programatically using the **signal()** system call..

Exercise 5 – Processes

1) Review the man page for ps and do the following,
   - display only your processes
   - display all processes
2) Identify some high priority tasks running on the system.
3) Identify some processor and memory intensive tasks.
4) Start a simple process of your own and experiment with running it in the background and bringing it to the foreground (see notes for example process to run).
5) Kill the process while it is running in the background.

Atlantic LINUX

Save the following scripts to files in your home directory using the following command

```
cat > <filename>.sh
```

```
-------------------------cut here----------------------------
#!/bin/sh
# loop1.sh
#
while [ 1 ]
do
        echo "hello $USER"
        sleep 5
done

-------------------------cut here----------------------------
#!/bin/sh
# loop2.sh
#
while [ 1 ]
do
        echo "hi $USER"
        sleep 5
done
```

Note that these scripts need to be executable by you before you can run them.

Working With Files

OS File differences

- EOL
- EOF
- Managing
  - ascii transfer mode
  - binary transfer mode
  - `todos` / `fromdos`
  - `dos2unix` / `unix2dos`
  - `tr` command

Linux/Unix systems, Windows/DOS systems and Mac systems all use slightly different conventions to signify the end-of-line (EOL) and end-of-file (EOF).

Linux/Unix systems use "\n" for EOL

Windows/DOS systems "\r\n" for EOL

Mac systems use "\r" for EOL

"\n" is also known as Line Feed (LF) and has an ascii value of 10.
"\r" is also known as Carriage Return (CR) and has an ascii value of 13.

Windows also uses a the SUB symbol (which has an ascii value of 26) to indicate an end-of-file (EOF). Linux/Unix systems and Mac systems don't use a special end-of-file character.

When transferring text files between different operating systems you need to be aware of these differences and manage them. There are lots of different ways of managing these differences. When transferring files using ftp – **ascii** transfer mode will translate files on the fly. Commands such as **tr** can be used to manually translate the file from one format to another (or transfer the files in **binary** mode and let the application handle the format). Linux distributions usually come with one of a number of utilities for converting between the different formats – commands include **fromdos** and **todos** or **dos2unix** and **unix2dos**.

Example using tr,

```
tr -d '\r' < file.txt
```

`find` – finding files on filesystems

- `find` command
  - searching on filenames
  - searching on file properties
    - type
    - age
    - size
    - permissions
  - doing something with what you find
    - exec
- `locate` command

The **find** command searches for files in a directory hierarchy. It can search on various properties of a file. The most basic search is against a filename, e.g.

```
find / -name ls
```

Searches the filesystem hierarchy that starts at / for all files called “ls”. Find supports wildcards, so to find any files whose name starts with “ls”,

```
find / -name ls*
```

More advanced searches are possible against such file properties as **type** (one of **block device**, **character device**, **directory**, **named pipe**, **regular file**, **symbolic link**, and **socket**), file **access** or **modification** time (**atime** and **mtime**), file **size**, file **permissions**, **owner** or **group** (**perm**, **user**, **group**) and many others. e.g.

```
find /var -type d
find /var/tmp -mtime +1
find /var/log -size +250k
find /home -user root
```

The default behaviour of find is to display any files it finds (**-print**) but it can also execute arbitrary commands on files it finds when **-exec** is specified,

```
find /home -type f -user root -exec ls -la {} \;
```

It can often be useful to pipe the output of find to the xargs command to do some processing (why are we not using wc directly?) e.g.

```
find / -name '*.c' | xargs wc -l
```

The **locate** command is also found on most Linux system. It is very fast as it queries a database of files on the system which is rebuilt periodically (usually overnight to minimise the impact on the system). The results of locate are dependent on how recently the database was updated.

grep - finding strings in files

- basics
- counting
- inverted matches
- searching many files

The **grep** command searches files for strings. In basic use, the command is passed a string to search for and one or more files to search in. It displays any matching lines it finds e.g.

```
grep Mozilla /var/log/apache/access.log
```

will search the file /var/log/apache/access.log for any lines containing the string “Mozilla” (but only “Mozilla”) and display those lines. Like most other Linux commands, **grep** is case-sensitive by default but you can over-ride this with the **-i** option. Using grep -i in the above example would also print any lines containing “mozilla”, “mOzIlLa” and so on.

If you want to count the number of matching lines without necessarily printing them out, the **-c** option can be used.

```
grep -c Mozilla /var/log/apache/access.log
```

will search the file /var/log/apache/access.log for any occurrences of the string “Mozilla” (but only “Mozilla”) and print a count of how many lines matched.

If you want to search for lines that **don't match** a particular string, you can use the **-v** option which **inverts** the match, so

```
grep -v Mozilla /var/log/apache/access.log
```

displays only lines which don't contain the string “Mozilla”.

grep can search many files at once by using a wildcard. If you are searching many files at the same time, the **-l** option may be useful to display only the names of files which contain a matching line, e.g

```
grep -l Mozilla /var/log/apache/*
```

Regular Expressions

- String matching patterns
- Easier to write than to read!

- Regular expressions consist of 2 parts

  (single character matches) + (repetition characters)

- Widely supported
  - Shell
  - Perl
  - Java
  - C/C++

Regular expressions are a more sophisticated approach to pattern matching for strings than that provided by wildcards. Regular expressions are supported, with minor variations, on various Linux commands and in a number of programming languages including Perl and Java.

Note that it is usually easier to write a regular expression than to read one so they do represent a potential maintenance problem – they are certainly very useful for Linux command-line activities though.

The basic building blocks of regular expressions are the patterns that match a single character. These are typically followed by a **repetition character**.

Single Character Regular Expressions

a	Most characters, including all letters and digits, are regular expressions that match themselves
.	matches any single character
[abcd]	The []'s match any single character between the brackets
[a-d]	The – in []'s matches any single character in that range
[^abcd]	The ^ in []'s matches any single character except those between the brackets.

Repetition Characters

?	Match the preceding character 0 or 1 times.
*	Match the preceding character 0 or more times
+	Match the preceding character 1 or more times
{*n*}	Match the preceding character exactly *n* times
{*n*,}	Match the preceding character at least *n* times
{*n*,*m*}	Match the preceding character at least *n* and no more than *m* times

Others

^	Match start of line
$	Match end of line

Regular Expression Examples

1. `a+`
2. `.*`
3. `[foo]+`
4. `Mo?zilla`
5. `[a-zA-Z]{2,5}`
6. `[Mozilla]{4,}`

Atlantic LINUX

1. Matches “a”, “aa”, “aaa” and so on.
2. Matches anything from “” (empty string) to “ASD£%$%:@”.
3. Matches “f”, “ff”, “fff”, “o”, “oo”, “ooo” , “fo”, “foo”, “ffoo” and so on.
4. Matches “Mozilla”, “Mzilla” but not “Moozilla”.
5. Matches “aa”, “Aa”, “of”, “foooo” but not “f” or “ffffff”.
6. Matches “Mozilla, “Mzilla”, “Moozilla” and “zool” but not “zoo”.

Can test these expressions using the following perl script (again, *cat > <filename>.pl* to save this to a file, make executable and run with *./<filename>.pl 'EXPRESSION'*).

```
#!/usr/bin/perl
#
#
use strict;
my @strings = ( "foo", "a", "aaa", "aa", "ff", "", "ffoo", "f", "o",
   "oo", "xiuy23193\$\|", "Mozilla", "Mzilla", "Moozilla", "zool",
   "zoo" );

my $pattern = $ARGV[0];
my $string;
foreach $string (@strings) {
  if ( $string =~ /^${pattern}$/ ) {
     printf("matched $string\n");
  }
}
```

Exercise 6 – Files and Regex

1) Create a text file on your Windows system and transfer it to the Linux system using ftp in ascii and binary transfer modes. Repeat using sftp.
2) Count the occurrences of the word "kernel" in all files starting with the letter "m" (hint: find and grep).
3) Write a single regular expression to match the following sequences in a file:
   - aaa
   - bbbb
   - bbb
   - aaaa

Atlantic LINUX

Other Useful Commands

Date and time

- `date`
  - format specifiers
- `cal`
- `time <command>`
  - real time
  - user time
  - system time
- `sleep <time to pause for>`

The **date** command **displays the current date and time** in a wide variety of formats. By default, the date command simply displays the current date and time in local time format. **Format specifiers** allow the output to be customised *e.g.*

```
date                    Thu Oct 28 23:18:27 IST 2004    (local time)
date +%d-%b-%Y  28-Oct-2004
date +%H:%M             23:18
date +%U                43                              (week of year)
date +%s                1099002149                      (seconds since epoch)
```

The **cal** command simply displays calendars (by month or year).

The **time** command has nothing to do with displaying the date or time. It is used to **run commands and display statistics about the system resources used** by the command while executing, including the **real** and **system time** it took to run. *e.g.*

```
# time grep testing /etc/*
/etc/lynx.cfg:# http://www.nyu.edu without testing www.nyu.com).
   Lynx will try to

real    0m0.067s
user    0m0.024s
sys     0m0.042s
```

The command took 0.067 seconds of real time to execute and the CPU spent 0.024 seconds in user code and 0.042 seconds in the kernel on behalf of the user code

The **sleep** command pauses for a no. of seconds, minutes or hours (useful in scripts).

More on viewing files

- Looking at the start or end of files
  - `head`
  - `tail`
- Monitoring files as they grow
  - `tail -f`
  - `less + SHIFT-f`
- File statistics
  - `wc`
- Sorting file contents
  - `sort`

Linux systems include a collection of commands for viewing text files in different ways.

The **head** command displays the first few lines of a file. The default is 10 but this can be changed with the **-n** option.

The **tail** command does the same job but from the bottom of the file. The **-n** option can again be used to control exactly how many lines are displayed. The ***+n*** option can be used to tell tail to start from the *n*th line in the file, rather than the last one. The tail command can also be used to monitor a growing log-file using the **-f** option. This can be useful when debugging a system.

As an aside, the **less** command provides the same functionality as **tail -f**. When viewing a file with **less**, you can start tailing it by pressing SHIFT-f.

The **wc** command is used to display statistics about text files including **line counts**, **word counts**, and **byte counts** of files.

The **sort** command is used to sort lines of text files in various ways including **by dictionary order** (**-d**), **case insensitively** (**-f**), **numerically** (**-g**) and so on. This is particular useful when combined with **redirection** and **pipes**.

Packing files

- tar
  - tar -cvf
  - tar -xvf
  - tar -tvf
- cpio
  - cpio -v
  - cpio -id

It can be useful to pack files for transport or archiving. There are two traditional tools used on Linux (and most UNIX systems) for packaging files. Archive files are normally named with an extension of the command used.

The **tar** command was originally used to archive filesystems to tapes. It can still be used for this purpose although there are more sophisticated tools for tape backups. It is normally used these days to create tar archives directly on filesystems.

To create a tar file

`tar -c -f <tar file name> <files/directories to package>`

To unpack a tar-file

`tar -x -f <tar file name>`

The **cpio** command is not commonly used any more although you may occasionally encounter cpio archives which need to be extracted, the following commands illustrate how to create and extract them for completeness. Note that these commands use I/O redirection and pipes.

To archive the contents of the current directory:

ls | cpio -v > <*cpio file name*>

To extract the files:

cpio -id < <*cpio file name*>

Compressing files

- `gzip`
  - standard
  - pretty good compression in short time
- `bzip2`
  - pretty standard
  - very good compression in medium time
- `compress`
  - traditional unix tool
- `zip` (or `jar`)
  - traditional dos/windows tool
- Combining `tar` with `gzip` or `bzip2`

As well as packaging files and directories into archives, it can be useful to compress them to save space.

The traditional compression command on Linux systems is **gzip**. The file to be compressed is passed as an argument and gzip generates a compressed version of the file with a **.gz** extension. gzip takes a number of options including ones which specify what kind of compression to use (including –**best** or –**fast**). Gzipped files can be decompressed using the **gunzip** command. Gzip uses Lempel-Ziv coding (LZ77).

More recently, a better compression tool has been introduced in the form **bzip2**. It is used in the same way as the gzip command and generates files with a **.bz** extension. Bzipped files can be decompressed with **bunzip2**. bzip2 uses the Burrows-Wheeler block sorting text compression algorithm and Huffman coding. It usually gives better compression (on the order of 10-20%) at a cost of more system resources and time compared to gzip.

You may occasionally encounter files with a **.Z** extension, these have typically been produced with the traditional unix compression command – **compress** (verify this with the **file** command!). They can be extracted by running **compress -d** on the file.

The **zip** format commonly found on Windows systems (and used in Java jar files) is also supported on Linux systems via the **zip** and **unzip** commands. Java's **jar** command uses the the zip file format so it can be used to extract normal zip files.

When using tar, the packing/unpacking and compression/decompression steps can be integrated with either **gzip** or **bzip2** using the **-z** and **-j** options respectively.

Scheduling

- Running jobs once at some future time
  - `at`
- Running jobs once when the system is under-used
  - `batch`
- Running jobs regularly
  - `cron`

```
0 7 * * * ~/bin/daily-backup.sh
0 7 * * 1 ~/bin/weekly-backup.sh
0 7 1 * * ~/bin/monthly-backup.sh
```


Linux provides a number of facilities to run commands at some time in the future without requiring user interaction at that time. The two main approaches use the **at** command and the **cron** facility.

The **at** command is intended to run commands once at a particular time, for example, if you wanted to start a large compilation during the middle of the night when it wouldn't inconvenience other users of the system.

An at job is scheduled using the *at time* syntax where time can be in a number of formats including *HH:MM*, midnight, noon, teatime, *month-name day year*, *MMDDYY, MM/DD/YY, MM.DD.YY*, *now + time* (where time is specified in units of *minutes, hours, days or weeks*).

The at command then prompts for the command(s) to be run and input is terminated with CTRL-D. Any output resulting from the command is emailed to the user. Pending at jobs an be viewed with the **atq** command. Jobs can be removed with the **atrm** command.

e.g.

```
$ at midnight
at> /bin/ls
at> CTRL-D
job 1 at Tue Mar 14 00:00:00 2006
```

The **batch** command is a variant of at which runs a scheduled job when the system load falls below 1.5

The **cron** facility is used to run recurring tasks on a periodic basis, for example, system backups are usually run through the cron facility. Jobs are scheduled by adding an entry to the users **crontab** file with the command *crontab -e*. The format of the file is

<minute> <hour> <day of month> <month> <day of week> <command>

e.g.

```
$ crontab -e

0 7 * * * ~/bin/daily-backup.sh
(exit editor)
```

Schedules the /bin/daily-backup.sh script to be run at 0700 every day.

Exercise 7 – Commands

1) Time the ls command and explain the various output fields.
2) Create a tar-file of your home directory.
3) Time both gzip and bzip2 compressing this tar-file and comment on the differences found.
4) Schedule a one-off job removing the compressed files in 5 minutes time.
5) Explain the following cron entries
   - `25 4 * * 1 w`
   - `* * * * * /bin/monitor.sh`
   - `0 * * * * /bin/wall /0.txt`
   - `5 9-17 * * 1-5 /bin/work`

Atlantic LINUX

Editing files

Introduction

- Typical editors on Linux systems
  - vi / vim
  - emacs / xemacs
- Starting vi
- Buffers
- vi modes
  - command mode
  - insert mode
  - switching modes
- Quitting vi

The **vi** text editor can be found on practically all Linux systems. There are more user-friendly and arguably more powerful text editors but anyone using Linux should be familiar with at least the basics of using vi. There are a wide range of vi-like editors all of which have the same basic characteristics.

To start using vi, invoke it at the command line as `vi filename`. If *filename* doesn't exist, vi will create it.

When you start vi, a copy of the file you are editing is placed into a **buffer**. The buffer is not written back to your file until you explicitly tell vi (details below).

The vi editor uses different **modes** to allow the user to perform different tasks. When you start the vi editor, it is in **command mode**. This is used to issue commands to the editor, move around the file, load and save files, quit and so on. To begin inserting text you must switch to **insert mode** by pressing "i". Most vi clones will display *–INSERT--* at the bottom of the screen to indicate the new mode.

When in insert mode, any keystrokes input are placed in the file. To switch back to command mode at any time press the *ESC* key.

Most commands in command mode start with ":". There are various options for exiting depending on whether you wish to save what you've input so far,

```
:q – quit vi
:q! - quit vi without prompting to save changes
:wq or ZZ or :x – save changes and quit
```

Navigation in vi

- Traditional navigation
- Cursor keys
- Paging
- Advanced navigation
  - start of line / end of line
  - next word
  - previous word
  - start of file / end of file
  - specifying particular lines

Navigation around a file is achieved while in command mode.

Traditionally vi has used the **h**,**j**,**k** and **l** keys to navigate around a file as follows:
h moves left one character
j moves down one character
k moves up one character
l moves right one character

Most newer versions of vi also support the cursor (arrow) keys for navigation.

You can also move through a file a page at a time using the traditional commands:
CTRL-F – move forward one page
CTRL-B – move back one page
CTRL-D move forward (down) one half page
CTRL-U move back (up) one half page

Page up and page down are supported on most modern vi clones.

There are various advanced navigation commands in vi also including:
0 – move to the start of the current line
$ - move to the end of the current line
w – move to the start of the next word
b – move to the start of the previous word
:0 – move to the first line of the file
:$ move to the last line of the file

To move to a specific line in the file, you use ***:n*** where n is the line number you want to jump to.

Cut and paste in vi

- Copy (yank)
  - single lines
  - multiple lines
- Paste
- Cut (delete)
  - lines
  - characters
- Inserting files

Cutting and pasting in vi is normal done in terms of lines. You can mark blocks of text within lines for cutting and pasting but it is normally easier to cut and paste the line(s). All text manipulating is done in command mode.

To **copy** (or **yank** in vi terminology) a line you use **yy**. Multiple lines can be copied by putting the number of lines before the yy e.g. **5yy** will copy the current line the cursor is on and the 4 that follow.

These lines can be **paste**d to the cursors current location using **p**.

Cutting (or deleting) is done with the **dd** command. Specifying a number before the dd again causes it to process multiple lines.

Deletion can be performed at a character level using the **x** command. Specifying a number before the x again causes it to process multiple characters.

Any deleted characters or lines can be pasted back to the cursors current position using the p command.

To insert the contents of another file into the current buffer, use the **:r *filename*** command where *filename* is the file whose contents you want to insert.

Search and replace in vi

- Simple search
  - repeating
  - backwards
- Simple find and replace
- Advanced search
- Regular expressions

Atlantic LINUX

To perform a simple text search of the current buffer use the ***/phrase*** command where *phrase* is the string you want to search for. vi will automatically jump to the next occurrence of that string. To continue searching from that point for the same string, press / again.

The search can also be performed backwards in the file by using the **?** command instead of / (***?phrase*** searches backwards for *phrase* and **?** on its own repeats the search for the previous phrase).

You can also do a **search and replace** on the contents of a file using the **:s** command. The syntax is a little more complex:

:s/*string to find*/*string to replace it with*/ (does a search and replace on the current line)

```
e.g.
:s/foo/blah/
```

:1,$ s/string to find/string to replace it with/ (does a search and replace from start to end of file)

```
e.g.
:1,$ s/foo/blah/
```

will replace each occurrence of the string “foo” with the string “blah”. The simplest form of the search and replace command only replaces the first occurrence of the string on each line it finds. To get it to replace all occurrences of the string on each line requires a slight change to the command,

:s/*string to find*/*string to replace it with*/g

Advanced searches can be performed by restricting the search to particular lines of the file.

Note also that both search and search and replace can use **regular expressions** for more complex matches.

Advanced vi

- Undo
- Starting editing on a particular line
- Replace mode

You can **undo** the previous command using **u** or **:u**. Some versions of vi allow multiple undos while others only allow the most recent change to be undone. An undone command can be redone with **CTRL-R**.

When editing a file with vi, you can automatically start on a particular line using the following syntax

```
vi +n filename
```

where *n* is the line number you want to start editing on.

Vi includes an alternative to **insert mode** in the **replace mode** accessed from the **command mode** with the **R** command. In this mode, any text input overwrites the existing text at the cursor.

Exercise 8 - Editing

1) Create a text file containing a few sentences of text.
2) Copy a few lines of the file.
3) Use search and replace to replace all occurrences of the word “the” in your file with the string “xxxx”.
4) Copy /etc/passwd to a file in /var/tmp and practice navigating around it.
5) Replace all non-letter characters in a copy of /etc/passwd with the letter 'X.

[Additional *Commands* exercise]

- Schedule a regular job using cron to back up your home directory to a compressed archive in /var/tmp. This job should run at 0500 every day except Sunday. Ensure that no-one else can view or extract the file.

Scripting

Introduction

- When to use shell scripts
  - system maintenance
  - batch operations
  - easily automated repetitive tasks
- When not to use them
  - webserver scripts (CGI)
  - high security jobs
  - heavily loaded systems
- Alternatives
  - Perl
  - Python

Shell scripts are a convenient tool for various system administration tasks. They are generally well suited to specific small tasks. As a general rule of thumb, if a shell script is exceeding a page or two, you should consider re-writing it in more sophisticated scripting language (like Perl or Python).

Shell scripts can be quickly put together and prototyped. Since they are interpreted they will always be slower than a compiled program (but are probably fast enough for most tasks). Large shell scripts can present a maintenance problem.

Do use scripts to automate system management tasks.

Don't use shell scripts on systems or for tasks where security is important – scripts are general not very secure. Remember also that anyone that can run a script can view the contents of that script so passwords or other secrets should never be stored in scripts (the same probably applies to most other interpreted and compiled languages).

Your first shell script

- #!/bin/sh
- Comments
- External commands
- Shell builtins
  - alias, bg, cd, echo, ...
- Shell constructs
  - if, for, while, ...
- Execute permissions
- &&
- ||

A shell script is just a collection of commands which you want to execute together. More complex shell scripts can use some of the programming language type constructs offered by shells.

The first line of any shell script needs to start with the **shebang - #!*<path to shell>*** this is a piece of *magic* used by Linux to figure out how to execute the script. A bourne shell script will start with **#!/bin/sh**, a bash shell script will start with **#!/bin/bash** and so on. without the #!, you would need to execute the script by passing it as an argument to the sh or bash invoked directly e.g.
sh script.sh

Since bourne shell programming is the most portable, the rest of notes and examples in this section will use bourne shell syntax. More powerful shell scripts that are slightly less portable can be created using bash. Shell scripts can be created with any of the other shells including tcsh but it is not generally recommended.

The # symbol is used for **comments**. On any given line, anything after a # is ignored and treated as a comment.

Any valid Linux command is usable within a shell script. The most portable shell scripts will only use basic commands which are guaranteed to be on practically all Linux systems. It is also generally recommended to use absolute paths to commands rather than rely on a users PATH being set correctly (alternatively, you can explicitly set a PATH at the start of the script).

In order to execute a shell script, it needs to have the execute permission bit(s) set. Unlike compiled programs, shell scripts must also have the read permission bit(s) set for any user that wishes to execute the script.

&& is a logical AND operator and || is a logical || operator. These can be used to ensure commands only run if prior commands also succeeded (with &&) or only if prior commands did not succeed (||).

cd /var/tmp/foo && rm * [why is this better than cd /var/tmp/foo; rm * ?]

hello worlds

```
#!/bin/sh
# this is a hello world script
echo “hello world”
```

```
#!/bin/sh
# this is another hello world
# script
/bin/echo “hello world”
```


Atlantic LINUX

Running a script

- #!
- Running scripts through the shell command
- Debugging scripts
  - -x
  - -v

Executable text-files with #!<shell> as their first line can be executed from the command-line. In addition, any shell script (with or without a #!) can be executed by passing it as an argument to a shell command e.g.

```
# bash script.sh
```

Debugging is also possible with shell scripts by invoking the shell with the **-x** or **-v** options (either in the #! line or by call the shell directly).

-v displays each line of the shell script as it is being run by the shell

-x displays each line of the shell and any arguments as they are being executed. The -x output is generally the most useful for debugging script

Example

```
#!/bin/sh -x
if [ date > /dev/null ]
then
  echo "the date command works!"
else
  echo "the date command doesn't work."
fi

# ./foo.sh
+ '[' date ']'
+ echo 'the date command works!'
the date command works!
```

Shell variables

- Setting
- Using
- Rules for naming
- All variables of type string

Variables an be used in shell scripts the same way they are used in interactive shells.

Variables are set using the same syntax as at the prompt
i.e.

```
VARIABLE=value
```

e.g.

```
foo=10
```

Be careful not to put spaces on either side of the "=". If the value needs to contain spaces, use quotes (") around the entire value.

Variables can be accessed by preceding the variable name with the $ symbol.
e.g.

```
echo $foo
```

would display

10

Valid variable names start with an alphanumeric character or the _ symbol followed by zero or more alphanumeric characters (do not use symbols such as $, ? or * in variable names). Variable names are case sensitive so *$foo* is not the same as *$FOO*.

Shell variables & quoting

- Single quote - '
- Double quote - "
- Back quote - `

When setting variables in scripts (and at the command-line), it may sometimes be necessary to quote the value as explained previously. The different quote character cause some different behaviors.

When a value is surrounded by **single quotes** (**'**), you are explicitly telling the shell to **not manipulate the string between the quotes**. e.g.

```
# foo='this is the value of foo'
# echo $foo
this is the value of foo
```

When a value is surrounded by **double quote** (**"**) the shell will **expand any variables or backslash sequences** in the string between the quotes e.g.

```
# foo='this is the value of foo'
# bar="this is $foo"
# echo $bar
this is this is the value of foo
```

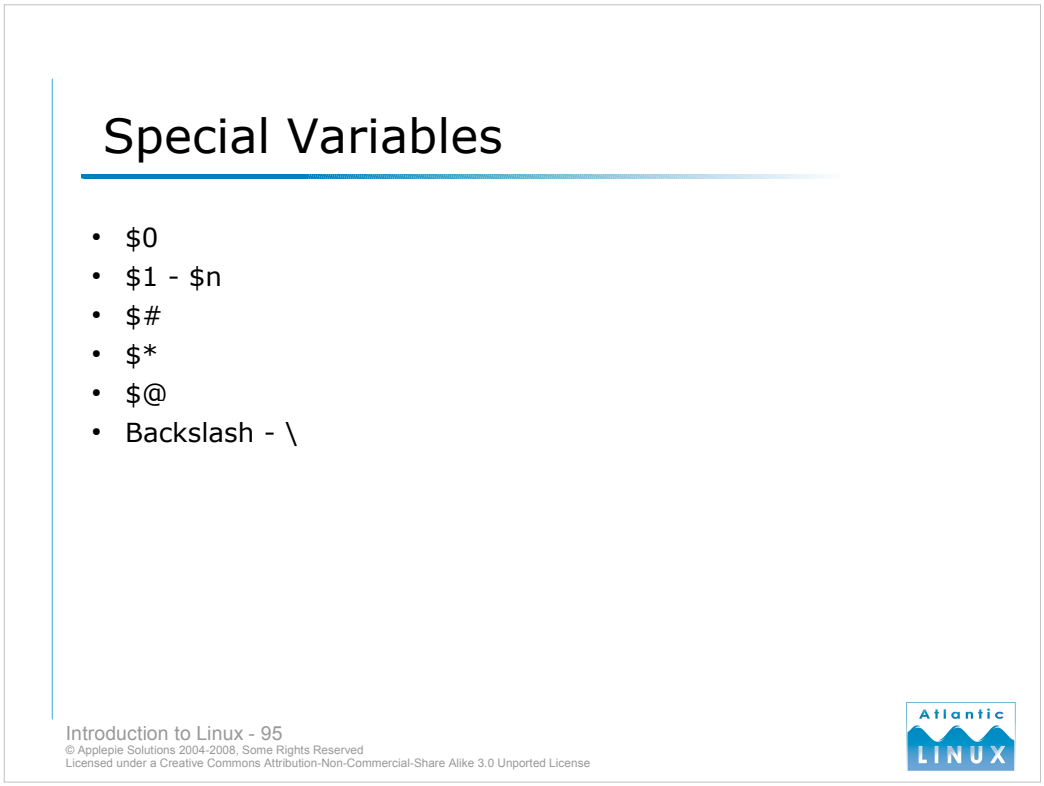
When a value is surrounded by **back quotes** (`) the shell will **attempt to execute the string** between the back quotes and replace the string with the output of the command. e.g.

```
# foo="the time is `date`"
# echo $foo
the time is Thu Nov  4 23:02:57 GMT 2004
```

There can sometimes be a performance improvement from using back quotes rather than other methods, contrast the time taken to run the following:

```
# time find /etc -name *.conf -exec grep foo {} \;

# time grep foo `find /etc -name *.conf`
```



As well as the standard special shell variables (such as $HOME, $PATH, $SHELL and so on), shell scripting in particular also uses a few special variables to handle arguments to shell scripts.

$0 always contains the full path to the shell script itself.

$1-$n contain the arguments to the script (some versions of sh may restrict n to 9 although bash doesn't have this limit).

$# contains the total number of arguments passed to the script

$*** and **$@ contain the same information in slightly different formats. The both contain the values for all arguments to the script. **$*** is a single string of all these values i.e. “$1 $2 $3 $4 ...” while **$@** is a list of all the values in double quotes i.e. “$1” “$2” “$3” ...

The **backslash character -** \ is also known as the **escape character** and is used to instruct the shell or a command to treat the character immediately after it as a normal character, even if it is normally a special character to the shell e.g.

```
# echo \$0
# touch \\foo
# mkdir this\ is\ a\ directory
```

Loops

- for VARIABLE in LIST
  do
  BODY
  done
- while EXPRESSION
  do
  BODY
  done
- until EXPRESSION
  do
  BODY
  done

The bourne shell supports variations of the loop constructs found in most modern programming languages. A common problem in loop constructs and shell scripts in general is missing whitespace where the shell is expecting it.

In the **for loop**, a list of strings is passed to the for loop and some action is performed on each of these items in the body of the loop.

```
for VARIABLE in LIST
do
        BODY
done
```

Newer versions of bash also supports a version of the for loop very similar to that used in the C programming language (but this won't be very portable).

The **while loop** continuously executes BODY while some EXPRESSION is true

```
while EXPRESSION
do
        BODY
done
```

The **until loop** is the opposite of the while loop, it executes the body of the loop until the expression is true.

```
until EXPRESSION
do
        BODY
done
```

The if statement

```
if CONDITION
then
  BODY
elif CONDITION
then
  BODY
else
  BODY
fi
```


The **if** statement is used to test one or more conditions, the basic syntax is

```
if CONDITION
then
        BODY
fi
```

More complex if statements can be constructed with a second alternative

```
if CONDITION
then
        BODY
else
        BODY
fi
```

Or with many options

```
if CONDITION
then
        BODY
elif CONDITION
then
        BODY
else
        BODY
fi
```

case and test

- case
- test (or [..])
  - STRING1 = STRING2
  - STRING1 != STRING2
  - INTEGER1 -eq INTEGER2
  - INTEGER1 -ne INTEGER2
  - -f FILE
  - ...

The **case** statement is an alternative to a chain of ifs and elifs. The syntax is very similar to that used in the C programming language and others

```
case EXPRESSION in
          ( PATTERN )
                BODY
          ;;
          ( PATTERN )
                BODY
          ;;
esac
```

In the case statement, if the EXPRESSION matches any of the PATTERNs, the related BODY is executed. It is common to use * for the last PATTERN to make a default action

A lot of these constructs rely on a true or false value to decide what action to take. Every command returns an EXIT CODE which can be used as a true/false value (exit codes are covered later on). We can also use the **test** command to test various conditions including string and integer equality, numerical comparison, the states of files (whether they exist, are a specific type, have a particular size and so on). the normal syntax is **test *expression***. An alternative syntax uses **[]**'s i.e. **[*expression*]**

When testing variables that may not have a value, it is recommended to use the following expression to avoid failures in test due to a missing variable,

```
if [ ${foo}X = "valueX" ]
```

where *value* is the expected string in *${foo}* and *X* is used to protect the test.

Exit codes,functions

- Exit status
  - checking
  - exit
  - $?
- ${VARIABLE}
- Functions
  - arguments
  - returning status

Every Linux command returns an *exit status* or *return code*. By convention, this code is 0 when the command has been successful and a non-zero value specific to the command when the command fails (in the range of 1-255). Scripts can use this both to test the return codes of commands they invoke (at the simplest level, a 0 return code is treated as true by test) and to return their own status code (using **exit <status code>**). The return code from the previously executed command is also stored in the special shell variable **$?**.

A significant error in shell scripts is failing to check $? after executing a command the output from which the script depends on to complete.

When referencing variables in shell scripts, it is sometimes important to be able to embed the variable in a string. When doing this, you need to indicate to the shell interpreter where the variable begins and ends. This can be achieved by surrounding the variable name with {}'s - $VARIABLE is the same as ${VARIABLE} so you can create scripts like,

```
echo "Results are stored in ${USER}_RESULTS.data"
```

Shell scripts can also be written to use rudimentary functions or procedures. The basic syntax is

```
FUNCTION_NAME()
{
        BODY
}
```

A function must be defined before it can be invoked (there are no function prototypes in shell scripts). Functions can also take parameters which use the same variables as shell script arguments ($1 .. $9). They can also return a status using the shell construct **return**.

Special devices

- Useful special devices
  - /dev/null
  - /dev/zero
  - /dev/random and /dev/urandom

The Linux filesystem contains a number of **special devices** which can be useful for shell scripting.

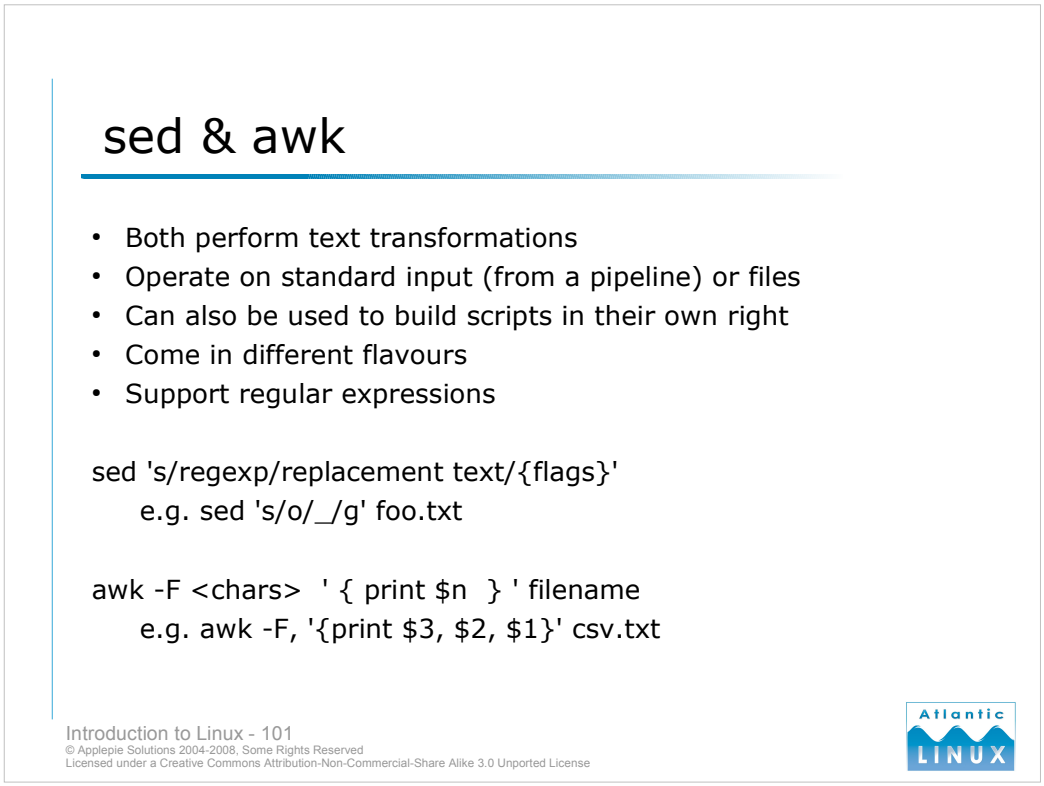
The **/dev/null** device is a data sink, any data written to /dev/null is discarded by the system. It can be useful in scripts to discard out the stdout or stderr of a command when executing it e.g.

```
#!/bin/sh
#
if [ date > /dev/null ]
then
  echo "the date command works!"
else
  echo "the date command doesn't work"
fi
```

The **/dev/zero** device is a source of *\0* (also know as NUL) characters. It can be used to feed dummy data to commands e.g

```
$ cat /dev/zero > zeros.txt
```

The **/dev/random** and **/dev/urandom** are a source of random data which be more useful than zeroes in some cases. The system generates random data using various sources of entropy on the system including network traffic and so on. The /dev/random device will only return random data if there is enough **entropy** on the system. The /dev/urandom device will always return random data but it may not be as random in low entropy situations.



Two other commands commonly used in shell scripts and **one-liners** are the commands **sed** and **awk**. Both commands can actually be used as scripting languages in their own right or can be incorporated into shell scripts.

They are powerful tools for transforming text in various ways (substitutions, re-arranging files and so on) and can take their input from standard input, a command pipeline or a file.

They support complex syntax and have a lot of overlapping functionality but a common use of sed is for performing text substitutions while awk is often used for selecting particular columns from text data.

Shell configuration files

- Bourne shells
  - /etc/profile
  - ~/.profile
  - ~/.bash_profile
  - ~/.bash_login
- C shells
  - /etc/csh.login
  - /etc/csh.cshrc
  - ~/.tcshrc
  - ~/.cshrc
- .bashrc (interactive and non-interactive shells)
- . and source

When a shell is started, it reads a configuration file which can be used to customise the environment and perform housekeeping tasks on behalf of the user. Typical actions performed might include customising the shell prompt and maybe starting a mail monitoring program such as biff. **Environment variables** can also be set in this file.

Each shell has their own particular configuration file (derivatives of the **bourne** and **csh** shells generally read either their own config file or their ancestors config files).

sh
- **/etc/profile**
- **~/.profile**

bash
- **/etc/profile**
- **~/.bash_profile**
- **~/.bash_login**
- **~/.profile**

csh
- **/etc/csh.cshrc**
- **/etc/csh.loginc**
- **~/.cshrc**

tcsh
- **/etc/csh.cshrc**
- **/etc/csh.login**
- **~/.tcshrc**
- **~/.cshrc**

Note that there are global configuration files under /etc. The shell typically reads these before reading the per-user configuration files in the user home directories. Environment variables such as PATH (and other system-wide variables) should be set in the files in /etc. These variables can be reset or extended by users in their individual configuration files.

The **source** or **.** commands can be used to read or re-read these configuration files.

Exercise 9.1 – Scripting

1. **Write a script which takes and processes the following options. The script should display an error message when the arguments to the script are incorrect,**
   - **-h displays a help message**
   - **-l performs an ls (with each line preceded by the command name)**
   - **-d displays the date in the form "22:34 25-Dec-2004"**
2. **Write a script which takes a filename as an argument and renames that file to an all lowercase version of the original filename.**
3. **Write a script that displays the following message**
   Hello *username*, today is *Day of week*

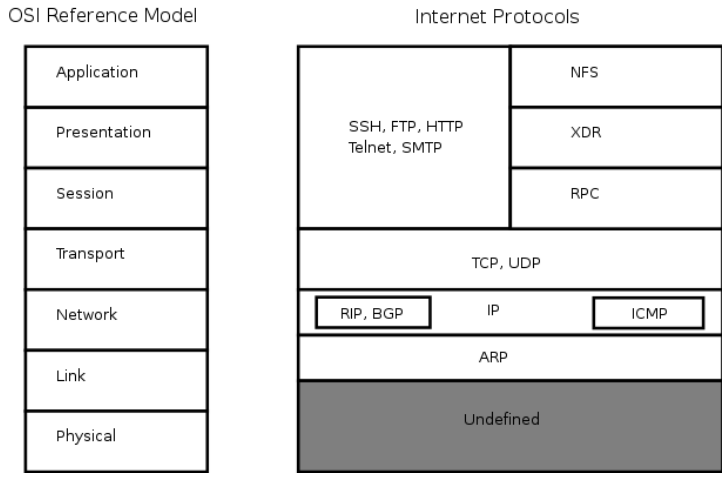
Exercise 9.2 – Scripting

4. **Write a script to back up any user's home directory ($HOME) using tar and gzip. Print a message and stop if it fails at any point ($?). Set permissions on the backup file so only the user can access the file. Name the file backup-*USERNAME-YYYYMMdd*.tar.gz**
5. **Write a script that takes two files as arguments (-m message.txt and -r recipients.txt). The script should email the contents of message.txt to each email address in recipients.txt and should also tell each user what time it is. Start each message with 'Dear *first name*'.**
6. **Write a script to add numbers e.g. add.sh 2 2 will display 4 on output (hint: expr and read).**
7. **Write a script to convert miles to km.**

Atlantic LINUX

Networking

Networking Concepts



Networks consist of **hosts** i.e. members, usually computers but possibly other participating devices such as printers. A collection of hosts on a network is sometimes referred to as a **site**.

All hosts on a network, as well as being connected by some medium either wired or wireless, must also agree on a set of protocols for communicating with each other. The OSI model is an abstract reference model for communications and computer network protocol design. The internet protocol suite commonly used on the internet roughly maps to the OSI model as shown in the diagram.

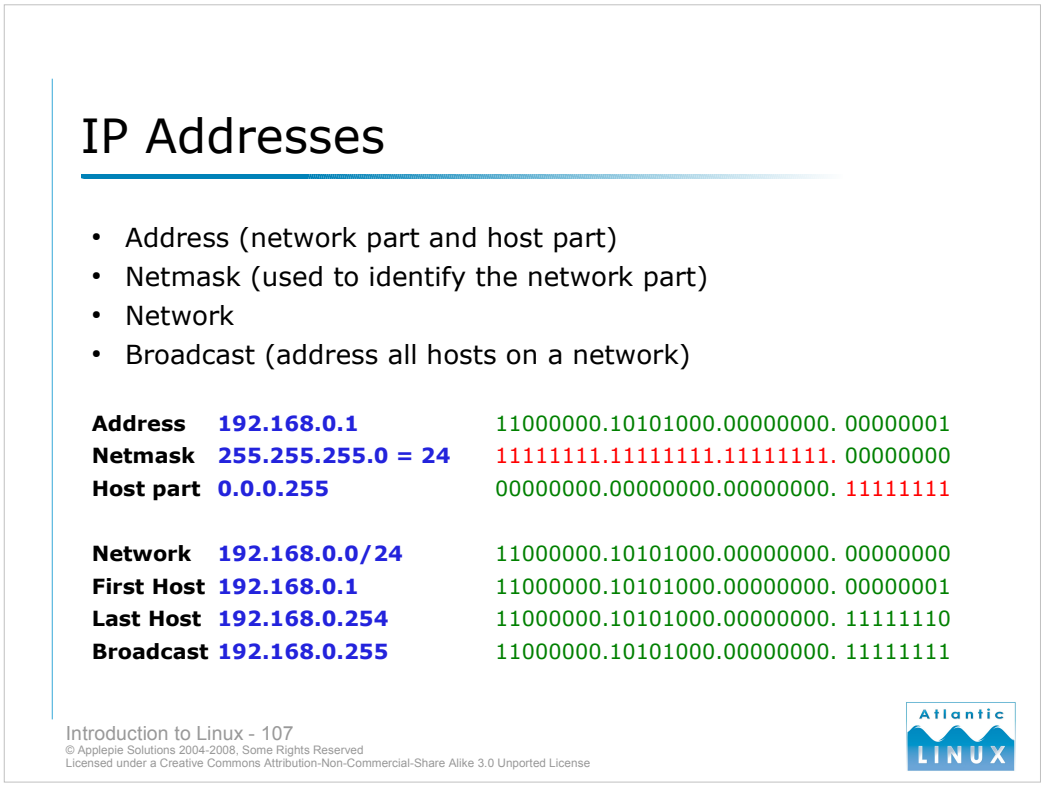
The everyday protocols you use are at the top (application layer). Examples include **ssh** (for connecting to machines), **ftp** (for transferring files), **http** (for browsing the web) and **smtp** (for sending email).

Underneath these applications, either TCP or UDP is used to actually transport data packets. Tranmissions Control Protocol (TCP) is a connection-oriented protocol that offers guaranteed delivery of packets (with an overhead in network handshaking that this requires) while User Datagram Protocol (UDP) is a connectionless protocol that transmits packets on a best effort basis without any guarantees about delivery.

TCP and UDP run on the Internet Protocol which describes the protocol used to encapsulate and transport data around the network. It handles details such as addressing and routing. Dynamic routing of IP packets around the network is handled by protocols including RIP, BGP and OSPF. Internet Control Message Protocol (ICMP) is used to send status and error messages around IP networks (this is the protocol used by tools such as ping).

At the lowest level, the Address Resolution Protocol (ARP) is used to map IP addresses back to the physical addresses used by the underlying hardware. In the case of Ethernet networks for instance, each network device has a unique physical address (the MAC address). ARP is used to map these addresses to the IP addresses used at the logical network level.

The internet protocol suite can run on a variety of physical networks and media including Ethernet, Token Ring and Wireless..



Networks consts of **hosts** i.e. IP identifies each host on the network with a 32-bit IP address (in IP v4), e.g

11000000 10101000 00000000 00000001

For convenience, these addresses are usually represented as 4 decimal numbers with each number mapping to a byte of the 32-bit address (ref: **ipcalc** command)
e.g.

192 168 0 1

This is usually split into a **network part** and a **host part**. So an example network might be 192.168.0 which contains a maximum of 254 hosts (192.168.0.1 to 192.168.0.254). 192.168.0.0 is the **network address** and 192.168.0.255 is the **network broadcast address**.

The netmask is used to identify the network part of an IP address.

The broadcast address for a network can be used to directly address all devices on a particular network e.g.

```
ping -b 192.168.0.255
```

sends an ICMP message to all active hosts on the 192.168.0 network.

Devices and Tools

- Network devices
  - eth0, eth1, ...
- Tools
  - ifconfig
  - ping
  - telnet
  - traceroute
  - route
  - ipcalc

Linux network devices do not appear in /dev. Network devices are created by the kernel device drivers when they recognise network hardware. They are named *eth[0..n]* in the order they are discovered by the operating system.

The **ifconfig** command is used for querying and configuring network devices – actions possible include assigning addresses and netmasks to devices, querying work devices and activating and de-activating network devices e.g.

```
ifconfig eth0 10.0.0.1 netmask 255.255.255.0
```

This would configure the first network device (*eth0*) with the address 10.0.0.1 and set the netmask to 255.255.255.0.

Running ifconfig with the **-a** option or with a specific network interface returns the current configuration for that device.

The **ping** command can be used to verify connectivity to a particular system on the network. It takes a hostname or IP address as an argument and attempts to send one or more test packets to that host. It reports whether or not it succeeds and provides statistics on the quality of the connection.

The **telnet** command can be very useful for testing text network protocols (as well as being a tool for connecting to systems) e.g.

```
telnet www.example.com 80
```

Connects you to the webserver port on www.example.com allowing you to send standard HTTP messages and review the servers response. In general, it is recommended to use ssh rather than telnet for actually connecting to systems due to telnets use of cleartext passwords on the wire.

The **traceroute** can be used to see the actual path across a network that packets are taking to a certain destination. It can be useful as a first step to diagnosing a routing problem.The **route** command is used to view or set routing information (for static routes).

The **ipcalc** command (*http://jodies.de/ipcalc*) is useful for testing network and netmask configurations.

Domain Name System - DNS

- Overview
- nsswitch.conf
- /etc/hosts
- /etc/resolv.conf
- host
- dig
  dig <hostname>
  dig -x <IP address>
- nslookup
  nslookup <hostname>
  nslookup <IP address>

The **Domain Name System** (**DNS**) is a system for mapping human-readable names to the IP addresses used by systems on the Internet. DNS misconfiguration is a potential source of problems and malfunctions in systems and system applications.

Linux systems use a number of sources for naming information – the system stops searching for DNS information when it finds an answer. The order in which a system searches for DNS information is controlled by the **hosts** line in */etc/nsswitch.conf*. It is normally configured as

```
hosts    files dns
```

This tells the system to look for DNS information in the **hosts file** first and, if that fails, to send a **DNS query** to a known DNS server.

The **hosts file** is a simple text file stored in */etc/hosts* which stores a list of IP addresses and hostnames in the following format,

```
IP_address canonical_hostname aliases
```

A **DNS query** involves sending a request to an external server (known as a **nameserver** or **DNS server**). The list of servers to query is stored in */etc/resolv.conf***.**

DNS queries can be conducted using either the **nslookup** or **dig** commands.

Exercise 10 – Networking

1. **Find the range of addresses for the 10.10.x.x network (using a netmask of 255.255.0.0).**
2. **Suggest a netmask and network for splitting a 10.10.10.x address into 4 separate networks.**
3. **Describe the fields in the network configuration of the first ethernet device on the Linux server.**
4. **Verify that the other server is running.**
5. **Identify the network path to that server.**
6. **Verify that there is a DNS entry for both servers (what is it?).**
7. **Verify if the mail service is running on the server (port 25).**

Atlantic LINUX

The System

The super-user account

- The `root` account
- Typical super-user activities
  - Filesystem maintenance
  - Software installation
  - Reviewing log-files
- `su`
- `sudo`
  - `/etc/sudoers`

Linux uses a dedicated user account for system administration – this user is usually known as **root** and has a **UID** of 0. The root account usually has access to change anything on the system. You should only use the root account when it is absolutely necessary, not for day to day activities. Be especially careful with the root account and the *rm -rf* command!

When performing root activities, you can either connected (by ssh or telnet) to the system as the root user or you can use the **su** command to temporarily assume the privileges of the root user. e.g.

```
auser> su
Password: *******
root>
```

The su command can actually be used to change identity to any other system user and can sometimes be useful for verifying the configuration of another system account (particularly when invoked as *su - <user>* which replaces your current environment with the user's environment).

An alternative to giving users full access to root privileges is the use of the **sudo** command. This allows a system to be configured to allow certain users to run certain commands as root without having full access. The users and commands they can run are described in the **/etc/sudoers** file. To invoke a command with root privileges a users calls *sudo command*. sudo also logs a lot of information about successful and unsuccessful attempts to invoke it providing a useful audit trail of root activities (see next slide for details about system logs). Using sudo also prevents lots of users from knowing the root password.

System log files

- syslogd
- /var/log
  - messages
  - syslog
  - apache/
  - ...
- Log rotation
- Reviewing
- dmesg

Linux systems log a lot of information about system activity to a series of log-files under **/var/log**. These log-files can useful when troubleshooting, reviewing system activity or tracking down system intrusions.

System logging is managed by **syslogd** which is configured with */etc/syslog.conf*. This file tells syslogd what messages to log and where to log them.

Most Linux distributions log all important messages to **/var/log/messages**. Additional information is often logged to other log-files under /var/log.

A lot of applications log their activity to separate log-files, applications such as the Apache webserver log their activity to files in either */var/log/apache* or */var/log/httpd* depending on the distribution.

A program called **logrotate** ensures that these log-files are archived and deleted periodically. It can be configured to rotate logs when they reach a certain age or a certain size (without log rotation, the /var/log partition would eventually fill up possibly causing system failures). Logrotate can be reconfigured by editing */etc/logrotate.conf*/.

System log-files should be periodically reviewed to identify any problems on the system. Basic tools including grep can be used to scan log-files for particular events or to review the behaviour of specific system daemons. System log-files can be monitored continuously with the **tail -f** command.

The **dmesg** command dumps the current output from the **kernel log buffer** – this usually contains the system boot messages but if there is a lot of kernel activity on the system it may be overwritten by newer messages.

Services

- Starting/stopping
- Adding new ones
- `/etc/init.d`
- Run-levels
  - `/etc/inittab`
  - `/etc/rc*n*.d`

When init starts, it starts any system commands that will be run in the background – these are called **daemons** or **services** and include commands like the system logger (**syslogd**), the internet super-server (**inetd**), the ssh service (**sshd**) and others. These are usually started with by a collection scripts in */etc/init.d*.

The scripts in **/etc/init.d** can be used to start, stop and check the current status of these services. New scripts can also be added in here if the administrator wishes to manually add new services to the system.

A Linux system can be started in various configurations. For example, if a system needs maintenance, it can be run in **single user mode** which prevents any other users from logging in and stops all non-essential services. Similarly, a system can be started in a multi-user configuration with no network services like webservers running.

These different configurations are specified by using what Linux calls **run-levels**. Linux has a number of standard run-levels (1 is single-user, 3 is full multi-user text mode, 5 is full multi-user graphical mode) and the system can be switched between the run-levels with the **telinit** command.

Linux systems contain a series of directories under /etc or /etc/rc.d called rc0.d, rc1.d, rc2.d and so on. Each of the directories contains **symbolic** links to the scripts in /etc/init.d specifying what services should be started and stopped for each run-level and specifying what order these services should be started and stopped. Red Hat provides the **chkconfig** tool for managing these or they can be modified manually. Scripts start with either “S” or “K” followed by a number. “S” scripts start a service, “K” scripts stop a service. The scripts are executed in increasing numeric order.

Software packages

- Overview
  - Red Hat (rpm)
  - Novell / SuSE (rpm)
  - Debian (deb)
  - Slackware (tgz)
- Dependencies
- Versions
- Package contents
  - Packaged software
  - Installation software
  - Package information
  - Dependency information

Linux systems contain a wide range of programs and applications. You can install new software on a system by downloading the source (when available) and compiling it yourself but this is time-consuming and can be a maintenance nightmare (especially when you want to move to a newer version of the program while preserving your original configuration).

Linux distributions have taken a number of approaches to **software packages** in order to reduce this maintenance overhead and simplify the installation and upgrading of software packages. There are two main formats used for packaging software on Linux systems – **RPM** – a format introduced by Red Hat (and now used also by SuSE, Mandrake and a range of others) and **DEB** – a format introduced by the Debian project (and used by various Debian derivatives including Knoppix, Ubuntu, Progeny and Xandros). The original "packaging format" used by some distributions (and still used by Slackware) – **TGZ** is simply an abbreviation of **tar.gz** which indicates the technologies used.

In all cases, the software package usually contains the following:

- information about the package
- scripts to manage installation, upgrades and deinstallation
- the software to be installed which can be either binaries or source code
- details of any libraries or other pieces of software that the package requires to operate correctly.

Software packages are typically compiled against certain **versions of libraries** for a **specific architecture** – package managers will only allow the package to be installed on a system that matches these requirements. Some package managers (particular the Debian ones) automate the management of these dependencies upgrading packages to required versions automatically if needed.

RPM

- Installing
  rpm -ivh <*package*>.rpm
- Listing
  rpm -qa
- Removing
  rpm -e <*package*>
- Advanced uses
  rpm -qi <package>
  rpm -qR
- Alternatives
  - yum
  - red carpet

Red Hat and SuSE use the RPM packaging system and the **rpm** command to manage packages. There are also various graphical tools for managing the packages on the system but it is useful to be familiar with the rpm command especially if you need to install 3rd party software on a Red Hat system.

- To install a new package, *rpm -i <file name>* is the minimum required. Typically, *rpm -ivh <file name>* is used to provide some more feedback on the installation process.
- If a version of the package is already installed, you can use the upgrade option instead of the install: *rpm -Uvh* which de-installs the other versions of the package after installing this version.
- To review what packages are already installed on the system, *rpm -qa* is used.
- To remove (erase) a package, use *rpm -e*.

The rpm command has a lot of more advanced features (the current man page for the rpm command runs to about 800 lines). The *-q* option can be used for more than just listing all installed packages, it can also retrieve information for individual packages including

-qi <package>	lists information about <package>
-qR <package>	lists dependencies for <package>
-ql <package>	lists all files in <package>
-qf <file>	lists the package which owns <file>
-q –scripts <package>	lists the scripts used by <package>

To run any of these commands against an uninstalled RPM, add *-p <file name>*

yum

Newer Fedora Core releases include a more advanced package management tool called yum. This provides advanced features such as automatic dependency management. Red Hat Enterprise Linux does not include the yum tool by default (but it can be made to run on RHEL), relying instead on the Red Hat Package Management Tool (next slide).

red carpet

Newer versions of SuSE provide a more advanced package management tool called red carpet. This is a graphical tool for installing new packages and updates from SuSE servers. It automatically resolves dependencies, selecting additional packages for installation from SuSE when required.

Exercise 11 – The System

1. **Write a script that sends an email to some address containing a list of todays log messages from the ssh daemon.**
2. **Write a script to remove all log-files that meet the following criteria:**
   - **older than 10 days**
   - **larger than 1 Megabyte**
3. **Check the status of the webserver service.**
4. **Check the current runlevel. List the services started and stopped for that runlevel.**

Atlantic LINUX

Developing on Linux

C on Linux

- Compilers
  - GCC
    - C, C++
    - Objective-C
    - Fortran
    - Ada
  - Intel
    - C, C++
    - Fortran (F77, F95)
  - The Portland Group
    - C, C++
    - Fortran (F77, F95)

Compilers

The standard C compiler used on Linux is the GNU C Compiler (GCC), versions of which can be used to compile C, C++, Fortran and other languages. There are also a number of commercial alternatives for specific platforms. GCC's main strength is its portability to different platforms, it is not noted for performance.

Commercial alternatives include,

- *Intel* sell high performance C, C++ and Fortran compilers which give better performing code in at least some cases when compiling code for Intel processors.
- *The Portland Group* sell high performance C, C++ and Fortran compilers for 32-bit and 64-bit x86 platforms (both *Intel* and *AMD* processors). The Portland compilers have a good reputation in the scientific community for HPC applications.

Users of architectures other than x86 may have the option of using a compiler from their hardware vendor.

Java on Linux

- JRE vs. JDK
- Installing
- What version of Java?
- GCJ
- Environment Variables
- IDEs
  - Eclipse
  - Netbeans

JRE is the **Java Runtime Environment**. This is used if you want to run java applications. It includes a **JVM** (**Java Virtual Machine**) and **run time libraries**.

JDK is the **Java Development Tool Kit**. This is used when developing java applications. It includes the **javac compiler**, **jar tool**, **compilation time libraries** and other components. JDK also includes the runtime environment.

Java is not packaged with most Linux distributions, but is available as a free download from Sun Microsystems. Download the corresponding an RPM or .bin binary file compiled for platform/os, and follow the installation guide provided.

Java is installed as an executable. Discover which java version is in your path using *java -version* command.

No environment variables are required to run java itself, but the following may be useful when running some java applications.

$CLASSPATH - used to define the directories and jar files to be searched for classes at runtime. NB: Rather than using CLASSPATH environment variable, java can be started with the -classpath option to explicitly define the search path.
$JAVA_HOME - often used to define where java is installed on the system.
$JDK_HOME - occasionally used to defined where the toolkit is installed.

There are a number of free IDEs available for Java development on Linux including **Eclipse** (http://www.eclipse.org/) and **Netbeans** (http://www.netbeans.org/).

Other scripting languages

- Perl
  - powerful text manipulation
  - commonly used for system administration tasks
  - syntax similar to shell and C
- Python
  - newer scripting language
  - more emphasis on OO
  - emphasises readability of code
  - uses indentation rather than curly braces to delimit blocks
- Ruby
  - very strong OO emphasis
  - intended to be easy to learn
  - *Ruby on Rails* - web application framework

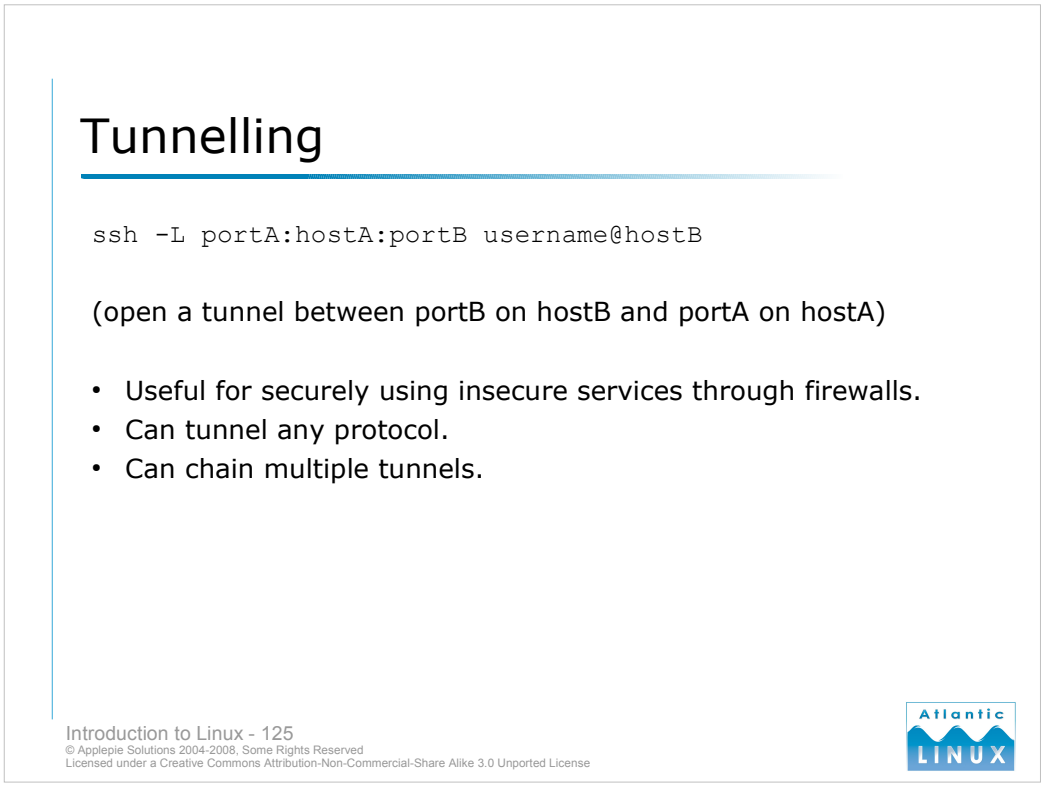
Exercise 12 – Developing on Linux

1. **Check what version of gcc is installed.**
2. **Write a simple hello world program in C and compile it up with gcc.**
3. **Check what version (if any) of Java is installed.**

Advanced SSH topics

Keys

- Uses
  - Eliminates passwords
  - Automation (remote commands)
  - Restrict certain commands to certain users and hosts
- HOWTO
  1) Generate your own set of keys
     (local host) `ssh-keygen -t rsa`
  1) Copy the public key to the system and account you want to access
     (local host) `scp ~/.ssh/id_rsa.pub user@hostname:.`
     (remote host) `cat id_rsa.pub >> ~/.ssh/authorized_keys`
  1) Verify you can login to that account without a password
     (local host) `ssh user@hostname`



Scenario

You have a network resource somewhere that you want to access (lets say a mail server or a webserver). There are multiple firewalls between you and it through which you only have ssh access (as the network administrators do not want to expose the mailserver or webserver to general access from the internet). You can use ssh tunnels to access the resource.

Example

System anoa is located on a private network which has a system called flame acting as a gateway to the internet. System beta is an intranet webserver on another private network which has a firewall/gateway system alpha. To access the intranet webserver on beta from anoa, we can create a series of SSH tunnels as follows,

Chain of SSH tunnels
anoa (9995) → (9995) flame (9995) → (9995) alpha (9995) → (80) beta
Virtual connection (over the tunnels)
anoa (9995) → → → → → → → → → → → → → → → → (80) beta

Note that each machine can only see the next machine in the chain, so anoa cannot connect directly to alpha or beta.

Procedure

1. Connect port 9995 on anoa to port 9995 on flame. The ports used are arbitrary, you don't need to use 9995 on both hosts.
   ```
   anoa> ssh -L 9995:127.0.0.1:9995 smulcahy@flame
   ```
2. Connect port 9995 on flame to port 9995 on alpha.
   ```
   flame> ssh -L 9995:127.0.0.1:9995 smulcahy@alpha
   ```
3. Connect port 9995 on alpha to port 80 on beta.
   ```
   alpha> ssh -L 9995:127.0.0.1:80 smulcahy@beta
   ```
4. On anoa, open a web browser and point it http://127.0.0.1:9995. You should see the output from beta's web server.

Syntax

```
ssh -L portA:hostA:portB username@hostB
```

means,

open a tunnel between portB on hostB and portA on hostA.

By default, only the localhost can connect to the portA. You can allow other hosts to connect to the forwarded port by using the -g switch to rss (or by ticking the box in putty "Local ports accept connections from other hosts".

In closing ...

- Summary
- Next steps
- Questionaire

Thank you and well done!