

Systems Administration - a boot camp.

A course for Systems Administration novices, with a Linux bias.

Copyright (c) 2006 Andy Davidson (Devonshire IT Limited)
andy@devonshire.it

LINUX is a registered trademark of Linus Torvalds.

Course Objectives

- Fundamentally, to teach good APPROACH and build on KNOWLEDGE.
- To build upon existing skills as a Linux user (not an introduction)
- To describe an accurate approach to problem solving
- To improve confidence with command line operations
- To solve problems with the creative use of scripting.

The objectives of the course are not to introduce Linux or Unix to the interested computer user, but to develop the skills of the established techie.

The course is designed to be as agnostic as possible; much of what will be taught will work on most Linux distributions, and our intention that as much of the worked examples and lessons in this course will be portable to other unixes as well.

The main focus however will be to teach people to administrate Linux systems, hosting network services.



SOME RIGHTS RESERVED

Released under a Creative Commons Licence

- Released under a Attribution-NoDerivs 2.0 UK: England & Wales
- You are free:
 - to copy, distribute, display, and perform the work
 - to make commercial use of the work
- Under the following conditions:
 - by Attribution. You must give the original author credit.
 - and No Derivative Works. You may not alter, transform, or build upon this work.
- For any reuse or distribution, you must make clear to others the licence terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder

- See <http://creativecommons.org/licenses/by-nd/2.0/uk/legalcode> for more information

In order to rapidly deploy this course, Devonshire IT Limited have used or redeployed some content which has been released into the Public Domain.

As a consequence, this licence has been adopted for the words, pictures, and format used in this course.

The material in this course is still Copyright - you are not allowed to make derivative works of this course without the the prior permission of Devonshire IT Limited, email <sales@devonshire.it> for more information.

Thinking in the correct way

Exploring the history of UNIX in order to understand what makes skills we learn performing one task, transferrable to all tasks we undertake on a UNIX system.

Handling changes.

I want us, in a few slides, to look very briefly at some of the standards which UNIX has been built against, in order to help us understand what we mean by portable, and compliant, and 'unixy'.

Thinking in a way similar to the original developers of UNIX will help you solve problems in the simplest and most portable way possible.

Principals of POSIX

- Thanks to POSIX we have applications which are:
 - Simple to learn (standard interface)
 - Predictable (signals, how a process starts, how to stop it)
 - Adaptable (pipes, standard inputs/outputs)
 - Safe (defined error handling)
- Thinking about how a process works in relation to these standards is key to improving the APPROACH of a new systems administrator - (e.g. pipes, exceptions).

POSIX is the Portable Operating System Interface for uniX.

The name for a family of rules. These rules define how compliant software interfaces with the unix-like Operating System.

This slide is a very basic overview. Don't recommend going to hunt out the standards and reading them!

POSIX is so important because understanding how to interface with POSIX compliant applications assists the administrator to support all POSIX compliant applications, and assists developers to make good choices when writing software. POSIX 1 is a standard that defines only very core features, but defines for instance, how a process is created and controlled, how signals can be thrown, and how to handle serious errors. POSIX 2 determines how a shell should behave (e.g. how to expand words/environment variables), and which builtin features a shell should have (e.g. kill).

A set of standards built into POSIX 2 define how a well behaved command will interface with the shell - which is why unix commands appear to have a fairly standard interface. Linux/Hurd/FreeBSD not officially POSIX certified, but conform to it.

For interest, the rules are now maintained by the 'Portable Applications Standards Committee' - <http://www.pasc.org/plato/>

Principals of UNIX

- Multi-tasking (a scheduler shares CPU time between processes) and Multi user (a process can be run under a different security regime from another process, and run in another environment)
- Everything can be treated as a file. File-system arranged in a single hierarchy (everything is under /)
- Concept of least privilege
- Unix derivatives have, in the past, shared many of their best features with other unices (NFS, TCP/IP support.....)
- Widespread use of plain text as the most portable storage mechanism.

UNIX was born in the 60s and 70s at AT&T's Bell Labs. Only UNIX derivatives which are certified as fully compliant with the 'Single UNIX specification' by 'The Open Group' are UNIX, but there are several 'Unix-Like' operating systems (e.g. Linux and Mac OS/X).

UNIX has many design principals, that when understood permit complex systems. UNIX is multi-tasking/multi-user, allowing many jobs on one server, each to be queued and processed fairly, under a different security regime. The file-system is arranged in a simple hierarchy, which allows administrators to abstract physical disk devices from data storage points, or mount points. Unix also allows us to treat everything (a device, a process) as a file.

System 5 UNIX defines much of how a modern UNIX functions. Referred to as 'SysV', many of the common features which we use today (e.g. SysV init scripts - /etc/init.d) were born at this time. The 'SysV Interface Definition' was another early effort to standardise good practice and behaviour for unix applications. Linux's past is traced through BSD, but most distributions borrow many features from the System V set. Solaris was derived through the SysV family. SysV Release 4 (SVR4) introduced (from BSD) support for TCP/IP, sockets, support for 'groups', and from Sun NFS, RPC and a better GUI. The primary platforms to install SysV SVR4 onto were the SPARC ... and the Intel PC. A UNIX for the masses is born.

Plain text leads to the most portable application (simple to configure, outputs and inputs can be shared between applications that the developers may never have heard of, easy to backup, good to compress, simple to develop against/script). However, it is hard to fit all alphabets into 7-bit ascii.

The UNIX philosophy

- Douglas McIlroy offers the universal summary of the UNIX philosophy
 - This is the Unix philosophy:
 - Write programs that do one thing and do it well.
 - Write programs to work together.
 - Write programs to handle text streams, because that is a universal interface."

When deploying software applications, and working on scripts, always remember that handling text streams is the most portable inter-process communication mechanism at your disposal.

Thinking Like a Systems Administrator

- Design Careful Procedures
 - Do make backups
 - Do change your passwords regularly
 - Implement change control (ensure you have knowledge, plan your change, test your change, make changes incrementally, make changes reversible test some more)
- Then actually follow your procedures.

Change control is always a hot topic, but it needn't be a complicated process. Following simple rules gives you a full back-out plan - when installing a new piece of software always take a copy of the distribution's default configuration file

```
cd /etc/asterisk
cp -p extensions.conf extensions.conf.dist
chmod a-x extensions.conf.dist
```

Then when making changes to configuration files, copy a known-good working file into a sensible place (often the same directory).

```
cd /etc/asterisk
cp -p extensions.conf extensions.conf.working20061131
chmod a-x extensions.conf.working20061131
```

Even better .. implement change control for config files with CVS or Subversion so that you can rollback config files to old versions, and account who makes a change when.

Formulating a plan to implement a change need not be an arduous task, and may only exist in your head for a simple change. That's better than no plan at all.

- You now know:
 - UNIX and UNIX-like operating systems share characteristics which make it easy to think in generic terms about all programs and processes.
 - When we confidently understand how one program throws exceptions, and returns successes and failures, we know how other programs will behave under the same conditions
 - A little about our approach to changing systems.
 - That the plain text stream is very important, and the ultimate enabler in terms of flexibility.



Files

We must assume that you are comfortable with creating files on Linux or unix, and that moving around the file-system is something you are familiar with. We will look at files from a systems administration point only.

Files are key to unix systems to an extent which simply is not the case for other operating systems. Access devices and files are so similar from the kernel's point of view, that they can both be configured - from a security point of view - in exactly the same, elegant way. Messages passed between processes on a system can be configured to pass through a simple file-like feature.

Under Linux, it is possible to tune operating system parameters and read system health data by interfacing with a simple text stream over a file presentation.

As a result, UNIX system security depends on a full understanding of protecting file entities.

File Ownership

- Files have a USER and a GROUP owner. The user that owns the file does not need to be part of the group which owns the file.
- Read, write, and execute privileges are assigned to every file (& directory)
- Group file ownership means granting access to a family of files to a user, is simply a case of adding the user to a group.
- Think about how this can secure applications/services as well as user documents.

A file's user and group ownership is completely de-coupled. One can build a very elegant and rapid to deploy file-service to users.

e.g. 'Tom' and 'Dick' are supervisors of the technical and commercial teams respectively in company X. The company file-server has groups 'sales' and 'tech', and all files used by each team, have their group ownership set to the correct team. Tom and Dick employ Harry, who is a technical sales person, and needs access to data produced by both teams. Making Harry a member of both groups permits him to see all of these files (where a read permission exists for the 'sales' or 'tech' groups) - without having to make a single change to the file ownership at all.

Think back to the first slides on this course where we looked at what a UNIX system is. The system is multi-user - meaning we can run lots of software in isolated, different, private environments, maintaining security between different applications running on the same box.

Consider an application which permits the exchange of data between your firm and other organisations. There are eight other firms that we want to exchange data with, and we use an identical software package to handle all communications. It is imperative that other firms do not see the files that we receive from other firms.

The application files which get run are owned by the 'exchange' user (rwx), and 'gateway' group (r-x). All of the third parties we want to exchange data with have their own account, which is a member of the 'gateway' group. All of these users can therefore read and execute the common executable files, but none of these users can read the files which are generated by interconnection with other firms.

Modifying Ownership

- Display ownership with `ls -l`
- Linux - can't give own files away (can on SysV)
- Initial group owner on most unices is primary group of user that created the file.
- Need 'w' on directory to create or delete file in that directory, but not to write to existing file (if user/group 'w' permission granted.)

```
andy@voipgw1:~$ ls -l
total 2376
-rw-r--r-- 1 andy users 757760 2006-11-27 12:17 cat
-rw-r--r-- 1 andy users 1667072 2006-11-27 12:17 dog
andy@voipgw1:~$ chown pop dog
chown: changing ownership of `dog': Operation not permitted
andy@voipgw1:~$ su -
Password:
root@voipgw1:~# chown pop /home/andy/dog
root@voipgw1:~# cd /home/andy
root@voipgw1:/home/andy# touch test
root@voipgw1:/home/andy# ls -l
total 2376
-rw-r--r-- 1 andy users 757760 2006-11-27 12:17 cat
-rw-r--r-- 1 pop users 1667072 2006-11-27 12:17 dog
-rw-r--r-- 1 root root 0 2006-11-27 12:45 test
root@voipgw1:/home/andy# chown andy:users test
root@voipgw1:/home/andy# ls -l
total 2376
-rw-r--r-- 1 andy users 757760 2006-11-27 12:17 cat
-rw-r--r-- 1 pop users 1667072 2006-11-27 12:17 dog
-rw-r--r-- 1 andy users 0 2006-11-27 12:45 test
```

File ownership is displayed with the `ls -l` command (third column is owning user, fourth column is the group owner)

Under Linux, a user cannot give their own files away, only the superuser can change ownership of files away. Under some SysV unices, a user can give their own files away.

Changing ownership of files handled by the `chown` command - `chown user:group file`. Can be done recursively with `chown -R`. More selective recursion handled with the 'find' command (e.g. change all directories only to be owned by 'diradmin' group 'diradmin' - `find . -type d -exec chown diradmin:diradmin {} \;`)

Change just the group ownership with the `chgrp` command.

Need 'w' (write) permission on a directory to CREATE or DELETE files in the directory. Can write to existing files with 'w' permission on file but not directory (more on this in the next slide)

File Access Types

	On file	On Directory
r	View contents	Search (e.g. ls)
w	Alter file contents	Create/delete files
x	Run executables	'cd' into it

- Setting file permissions in /dev affects access to devices.
- Changed with chmod command (chmod u+rw, chmod g-w, chmod o=rx, chmod o=)

ls -l's second column determines what the access types are. Look like : -rwxr-x--- - this file can be read and written to and executed by the user who owns it, and read and executed by members of the group that owns the file. If you are not the owner or in that group, you have no permissions to read from or write to the file.

You may be able to search the file (i.e. see it in an 'ls' on the directory if you have r and x permissions on the directory.) Specifically you need 'x' permissions on a directory to make it your current directory (cd into it), but you can handle the contents (read files, modify them, run them) of a directory with just 'r' permissions. Write permissions on a file are not required to delete it, if you have write permissions on the containing directory. Setting just 'x' on a directory allows users to work with programs in that directory that they already know about, but hides everything else.

Can control access to devices (e.g. line printer, modem, serial port) with file permissions - to make user andy able to send to modem :

```
# ls -la /dev/modem /dev/ttyS1
lrwxrwxrwx 1 root root    5 2005-11-09 21:06 /dev/modem -> ttyS1
crw-rw---- 1 root uucp  4, 65 2006-03-21 19:00 /dev/ttyS1
# adduser andy uucp
```

chmod command well defined in man page. chmod <set>=<nothing> removes all privileges. Specify defaults for environment (e.g. user session, or operating environment of a system service) using the umask command - see man page.

Special unix permissions

- Sometimes instead of 'x' in the execute position, an administrator sets a different execute permission.
- 't' - Sticky bit (files) - leave program in memory after it executes. the 't' letter comes from the phrase 'save text mode'. No start-up overhead for big tasks.
- Sticky bit on directories means 'user may only delete files that he owns, or where he has write permission on file, even when user has write perms to the directory (e.g. /tmp)
- 's' - SUID/SGID - executes the file with the process ownership (either user or group ownership) set to the user/group ownership of the file.
- SUID enormously powerful but DANGEROUS (chmod u+s /bin/bash)

Running programs SUID root is not uncommon, but should not be undertaken lightly. Imagine the consequences of unintentionally setting the perl interpreter to suid root - anyone who can write a perl script can now run commands on your system as the owner of the perl executable (which is likely to be root for the system-wide perl interpreter.)

Imagine the consequences of setting the bash shell suid root. What will every single user then have at their disposal ?

Many programs sometimes need setuid privileges - e.g. how else can a user change their own password ? The key is that it TEMPORARILY allows a normal user to do 'normal' things which required elevated privileges, in a controlled 'jailed' environment.

Directories with SGID permission set, force all files and subdirectories under this directory to be group-owned by the owner of the directory, rather than the default group of the user creating the file.

Managing the UNIX File-system

- A 'file-system' is a method of storing computer files, to make it easy for the OS to find them.
- All storage devices, e.g. CDs, Disks, when used to store computer files need a file-system to organise/index the files on disk.
- Sometimes file systems are 'fancy' - /proc, /dev
- Sometimes file systems are hosted on remote computers - NFS, CIFS
- Think of a file-system as a database of files

A file-system is a collection of files, and information about how to retrieve the files.

Some file-systems are fairly unique to the unix-like OSes - e.g. UFS, ext3. Some are common to many OS families (e.g. Joliet for CD ROMs, etc..)

If we agree that a file-system is a collection of files, and how to find them, then if a device is a file in Unix, as is a process, then there's a file-system to deal with the way that we interface with these file-like entities - /proc's filesystem is not like ext3, but it is still a filesystem. In this respect a file-system need not be attached to a storage device.

A network file system acts as a client for a remote server resource.

A storage device is not typically referenced directly - normally the file system sits between the files you want to manage, and the disk itself. The files available on the storage device must be made available via what is termed a 'mount point'.

Sometimes storage (devices like CDs, SD cards or over a network) is desirable only to mount when a user or program wants to access this data. automounting or supermounting facilitate this. With autofs/automounting, a filesystem is mounted transparently (to the end user) when a resource is first requested. This is a resource saver on network file servers.

fstab

- Storage<>mount-point mappings defined in /etc/fstab

```
# /etc/fstab: static file system information.
#
# file system<mount point> <type> <options> <dump> <pass>
proc /proc proc defaults 0 0
/dev/sda1 / ext3 defaults,errors=remount-ro 0 1
/dev/sdb1 /home ext3 defaults 0 2
/dev/sdb2 none swap sw 0 0
/dev/sdb2 none swap sw 0 0
/dev/hdc /media/cdrom0 udf,iso9660 user,noauto 0 0
```

- Defines where the storage engine is, where to put it in the hierarchy, the filesystem driver to load, and any extra options.
- 'dump' option adjusts archive scheduling rules.
- 'pass' option - 1 means do fsck on mount, 2 is default, 0 means do not fsck.

Disk IO taking a long time can cause a server or application to feel poor. It is therefore essential to think carefully about how disk IO is managed.

Valid options include :

- auto/noauto - whether or not to mount the filesystem at boot or with a mount -a (you may want to explicitly mount)
- exec/noexec - can binary execution occur from files on this volume (security feature)
- suid/nosuid - permits or disallows suid functionality (security feature)
- ro/rw - read only or read & write
- user - permit any user to mount this fs - implies noexec, nosuid, nodev by default.
- noatime - do not record file access times (improves performance at feature loss trade off)

defaults are rw,suid,dev,exec,auto,nouser,async.

The 'mtab' file lists the currently mounted file-systems. This should not be edited by an administrator, as applications such as 'mount' will take care of this file.

inode

- Regularly encounter the phrase 'inode'.
- An inode is a data structure (efficient method to store data) that stores information about one file, one directory or other file system object.
- Each file has an inode and is identified by a unique inode number. Inodes store information about user/group ownership, and permissions (access mode).
- There is a fixed number of inodes, so hence a fixed number of files or directories on a disk. Inode number can be seen with `ls -li`
- "hard link" can only be between two files on one physical volume - as its simply several 'locations' of a file pointing to the same inode.

An inode is represented as a positive integer. All files are links to an inode. Whenever the OS refers to a file, it is transparently mapped to an inode number.

A hard link is just one inode, referred to by several 'files'. e.g.

```
mkdir /test
touch /test/1
ln /test/1 /test/2
ln /test/1 /test/3
```

1, 2, and 3 do not show as links in `ls`, they are filenames which point to the same inode on the disk. The 'reference count' of this inode is 3. If one of these 'files' is deleted, the reference count will be 2 (two 'files' pointing to the inode.) When the reference count reaches 0, effectively the file is deleted. Conceptually, a unique file on a disk is just an inode with a reference count of 1.

A symbolic link is a reference from one file to another file, but at the filename level, and not the inode level. Symbolic links can be used across file-systems. (e.g. if `/usr` and `/usr/local` are different partitions, a symbolic link from a file on one to a file on the other will still work.)

Historically referred to as an i-node (the i possibly standing for index ... no-one knows.)

Compare typical Linux file-systems

	Good for	Bad for	Largest file	Largest volume	Num Files
ext2	Ubiquitous. Transparent compression option Undelete possible.	No Journal	2TiB	16TiB	10 [^] 18
ext3	Journaling. Can fail back to ext2 if journal fails. Can upgrade from ext2 without wiping data from disk. Easy to recover. Online filesystem growth.	No variable block length. Can not be repaired when mounted. Can not cluster. No defragmentation tools.	2TiB	32TiB	Varies
XFS	Oldest Journalled f/s, so very stable. Improved parallelism on large volumes through internal partitions esp. on SMP, stripe functionality on raid vols. Can grow filesystem dynamically. Online defrag. Excellent support for enormous files/vols.	No snapshot functionality, Can't shrink xfs dynamically. Grub <0.9.1 no support. No undelete possible (can be good). No Windows support. Journals only metadata, not files. Relatively new to Linux	9EB	9EB	No Max
Reiser	Online growth, Fast when dealing with small files.	No sync on directories - race. Enormously unstable in early days.	8TiB	16TiB	4bn
GFS	Clusterable. All nodes are peers.	Non-RedHat Linuxes.			

Only an introduction. There are lots and lots of file systems, many have different purposes beyond 'just what is available in my OS' !

Consider what you want features you need in a filesystem (clustering/startup time/large files..).

A journalling filesystem stores information about changes to disks so that in the event of a power failure, changes to disks can be replayed, reducing the risk of data loss in a failure condition.

Parallelism means 'doing lots of things at once' - and leads to a performance win, if there is no overhead associated with simply running processes in parallel.

A note about undeletion - it's better from a security point of view to not be able to undelete files, but undelete might help if you make a mistake !

Dynamic inode allocation allows a filesystem to allocate inodes based on need - frees up space when it is no-longer required, eliminating maximum numbers of files per filesystem.

Disc Misc

- Disks filling up is a common problem.
 - try to monitor your servers so that you prevent this from happening
 - df shows disk partition utilisation (df -h is easier to read)
 - du shows utilisation per directory (du -hs * is easiest to read)
 - Don't let your disks fill up - causes fragmentation problem.
 - ext3 reserves some blocks for superuser exclusive use - tune2fs to change.
- Quotas help restrict the amount of disk space available on a per-user basis.

```
andy@joy:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1       143G  2.4G  133G   2% /
/dev/sdb1       143G   46G   90G  34% /home
```

```
andy@joy:~/backups$ du -hs *
7.7M    letter
20M     number
```

```
andy@joy:~/backups$ sudo tune2fs -m 0 /dev/sdb1
tune2fs 1.38 (30-Jun-2005)
Setting reserved blocks percentage to 0% (0 blocks)
```



Processes

We now know enough about files to securely administrate a Linux system from that point of view. A unix system is not terribly useful if we can not run applications on it.

A process is any single program in execution. Processes come from many places - the kernel may cause 'something' to run (e.g. bdflush - synchronise changes to filesystem onto the filesystem), or perhaps a user has caused a program to run (the bash shell, or the apache web server.)

A 'command' may be one process, or several processes. A simple command like 'cat' causes one process to run, until the process has finished. A command which cats a file, and pipes it into another application for processing, and then maybe pipes it into another command to send the output to an administrator runs as three processes, one process per pipe segment.

The life-cycle of a process

- Always three system calls - fork(), exec(), wait().
- Consider how you interact with a command line
 - You enter a command, e.g. cat filename
 - The shell fork()s. A child process is created.
 - The child runs, the shell waits for the child to die.
 - When the child exits, it wakes the parent which has been wait()ing.
- A parent/child relationship between processes always exists. The parent should always outlive the children in UNIX. The parent 'sweeps' for exited children with a wait() call, when it learns the child is finished, and the parent learns the exit signal. If the parent process dies, init (process id 1) becomes the new parent and wait()s for the child to die.

The parent/child relationship is imperative in unix, in order to spread messages about a process's return condition ("did it work").

A child process will inherit the environment of its parent under normal circumstances (ownership, environment variables)

ps tree on a linux box shows this relationship

```
init--+-6*[agetty]
      |-screen--+-bash---asterisk---asterisk---26*[asterisk]
      |           `--bash
      |-sshd---sshd---sshd---bash---bash---pstree
```

Edited just to show two of my logins. ps tree's parent process is bash, who's parent process is sshd who's parent is init

For the very very interested, a SIGCHLD signal is often passed from the dying child process to the parent to cause a wait() system call to run, and collect the exit status.

Process priorities / The Scheduler

- Most Unix/Unix-like OSes do not have a real-time scheduler.
- Linux does not - uses a time-sharing algorithm to ensure all processes get some time in which to run.
- Processes can be given more (or less) priority (i.e. a larger slice of the time) with the nice command, e.g.

```
$ nice 10 make
```
- Running processes can have priority altered with renice

```
# renice -10 -p 1000
```
- Higher numbers are worse priorities. Superuser can give negative numbers (better priorities - as with the renice example above.)

Scheduling is not real-time, and tends to be interrupt based and time shared. All unix processes, no matter how insignificant, are carved up some time to function in.

In the example above, the 'make' command here will run with a relatively low, or worse priority. The process id 1000, in the second example, will start to receive more priority.

A process that 'recently' needed more cpu time will be given more priority by default, than one that does not. The scheduler contains many features like this in order to make the most efficient use of the CPU.

A thread is a little like a process - it is a cluster of processes which share the same code or data. Starting one process lots of times does not create a threaded application, but starting a process which creates lots of threads, sharing the same code and running environment does.

Job Control

- Many shells offer job control
- CTRL and Z to stop the running process
- & at end of command line to force it into the background
- 'jobs' (in bash) to list
- bg/fg [number] to move stopped or running job into foreground or background

```
make[2]: Entering directory '/usr/local/src/asterisk-1.4.0-beta3/menuselect'
gcc -Wall -o menuselect.o -g -c -D_GNU_SOURCE menuselect.c

[1]+  Stopped                  make
root@voipgw1:/usr/local/src/asterisk-1.4.0-beta3# sleep 60 &
[2] 10568
root@voipgw1:/usr/local/src/asterisk-1.4.0-beta3# jobs
[1]+  Stopped                  make
[2]-  Running                  sleep 60 &
root@voipgw1:/usr/local/src/asterisk-1.4.0-beta3# bg 1
[1]+  make &
root@voipgw1:/usr/local/src/asterisk-1.4.0-beta3# gcc -Wall -o menuselect_curses
```

Job control is very useful and a feature in many shells including bash.

Can be useful to put a program into the background when you have already started it.

Often use it to dive into and out of my full screen editor, vi. CTRL+Z in vi to drop me to a terminal, check something, then fg to go back into vi.

Scheduling jobs

- Run a command once in the future with 'at'
- Run a command periodically with cron
(see man 5 crontab for info)

```
andy@joy:~$ echo "echo \"it's 4.15pm - hurrah\"" | at -m 16:15
job 1 at Thu Nov 30 16:15:00 2006
```

```
andy@joy:~$ atq
1 Thu Nov 30 16:15:00 2006 a andy
```

```
crontab files :
<min> <hour> <dom> <mon> <dow> <command>
0 0 * * 1-5 echo "its midnight on a weekday"
```

edit with `crontab -e` - never edit the cron files in `/var` by hand - these are artefacts of the edit.

Make sure your clock is right ! `ntpdate pool.ntp.org` to sync periodically - good idea to schedule.

ps

- “What programs are running?”

- `ps -ef` vs `ps aux`

- Shows PID (unique process ID), who is running the process, and the state of the process.

- Is typical to panic when observing zombie processes. There may be nothing wrong - all processes Zombie when they exit normally.

```
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.1  696  244 ?        S    Nov19 0:06  init [3]
root         2  0.0  0.0      0     0 ?        S    Nov19 0:00  [kexentd]
root         3  0.0  0.0      0     0 ?        SN   Nov19 0:00  [kcoffload_CPU0]
root         4  0.0  0.0      0     0 ?        S    Nov19 0:00  [ksvcpd]
root         5  0.0  0.0      0     0 ?        S    Nov19 0:00  [doffish]
root         6  0.0  0.0      0     0 ?        S    Nov19 0:00  [kupdated]
root         7  0.0  0.0      0     0 ?        S    Nov19 0:00  [ahc_dev_0]
root         8  0.0  0.0      0     0 ?        S    Nov19 0:00  [scsi_ah_1]
root         9  0.0  0.0      0     0 ?        S+   Nov19 0:00  [atrecoveyrd]
root        10  0.0  0.0      0     0 ?        S    Nov19 0:00  [kjournald]
root        61  0.0  0.4  1428  592 ?        Ss   Nov19 0:00  /usr/sbin/syslogd
root        64  0.0  0.3  1384  452 ?        Ss   Nov19 0:00  /usr/sbin/klogd -c 3 -x
root       164  0.0  1.1  3280 1588 ?        Ss   Nov19 0:07  /usr/sbin/sshd
root       172  0.0  0.4  1576  620 ?        S    Nov19 0:00  /usr/sbin/crond -l10
root       176  0.0  0.3  1376  480 tty1    Ss+  Nov19 0:00  /sbin/getty 38400 tty1 linux
root       177  0.0  0.3  1376  480 tty2    Ss+  Nov19 0:00  /sbin/getty 38400 tty2 linux
root       178  0.0  0.3  1376  480 tty3    Ss+  Nov19 0:00  /sbin/getty 38400 tty3 linux
root       179  0.0  0.3  1376  480 tty4    Ss+  Nov19 0:00  /sbin/getty 38400 tty4 linux
root       180  0.0  0.3  1376  480 tty5    Ss+  Nov19 0:00  /sbin/getty 38400 tty5 linux
root       181  0.0  0.3  1376  480 tty6    Ss+  Nov19 0:00  /sbin/getty 38400 tty6 linux
root       6859 0.0  0.0      0     0 ?        S    Nov22 0:00  [khubd]
root       6923 0.0  1.4  3664 1792 ?        Ss   Nov22 0:05  SCREEN
root       6924 0.0  1.3  2724 1656 pts/1  Ss   Nov22 0:00  /bin/bash
root       8489 0.0  1.3  2720 1656 pts/4  Ss+  Nov22 0:00  /bin/bash
root      29661 0.0  6.7 18556 8464 pts/1  S+   Nov23 0:00  asterisk -vvvvvvvc
root      29662 0.0  6.7 18556 8464 pts/1  S+   Nov23 0:01  asterisk -vvvvvvvc
root      29664 0.0  6.7 18556 8464 pts/1  S+   Nov23 0:00  asterisk -vvvvvvvc
```

ps shows you what programs are being executed. On a BSD derived system, use `ps -aux`, on a SysV type system, use `ps -ef`.

The ‘STAT’ column is very useful for debugging. S is sleeping. R means on the run queue (“i am running or want to run”). D means that the process is asleep and can not be interrupted (often can be seen when trying to write or read from a faulty, or highly loaded device, e.g. disk), Z means Zombie

A person who has become a Zombie can not be killed, ergo a process which has become a zombie can not be killed. A zombie process is already dead. All processes which die become zombie processes eventually, this is normal and intended behaviour. The process has completed execution, and is waiting for the parent to find it in a `wait()`, and read its exit status.

A process which remains in the Z state for some time is indicative of a problem in the parent process. To clean up a zombie process, you could try sending `SIGCHLD` to the parent process with the `kill` command, or remove the parent process, causing ‘init’ to adopt the zombie’d child process. A zombie process which sticks around wont cause untold damage, but again do point to a leak of some kind in the parent technology.

init

- init is the first process (always PID1). Acts as an orphanage for processes who lose their parent. At boot, init creates a temporary shell to restore the system, and create daemon processes (e.g. servers)
- init finally creates 'getty' processes on each destination that needs an interactive login (inittab - virtual terminals & serial port = normal destinations).
- The getty calls the login program as a child process.
login verifies the credentials supplied, and if valid change to that of the user
login spawns the desired shell, e.g. bash.
The shell has its own children.
When the user logs out, all return codes as passed back up the chain to init.
init respawns a new getty.

The role of init is very simple.

Sometimes a parent will deliberately orphan a child at birth, e.g. processes started by gnome, etc. This means the parent does not need to check for exits.

init is normally the parent process of all daemon processes. Daemons simply exist to do a job (serve web-pages, receive emails). Daemons have no terminal attached, and sleeps for much of the time (e.g. waiting for an inbound connection attempt).

Signals

- Signals are messages that can be passed bidirectionally, and originate in the kernel or another process.
- Some signals are errors, some are informational.
- Regular error signals which are used :
 - 1 SIGHUP (Hangup)
 - 2 SIGINT (Interrupt (e.g. CTRL + C or 'DEL' key on SysV))
 - 3 SIGQUIT
 - 9 SIGKILL
 - 11 SIGSERV (Segmentation fault)
 - 13 SIGPIPE (Broken Pipe)
 - 14 SIGALRM (Alarm Clock)
 - 15 SIGTERM (kill)

Signals are passed to processes for information or in the event of an error. An example of an informational signal is the 'virtual terminal bell' which can be caught and ignored or cause something to happen, e.g. the screen to flash.

You can ask a process to quit by sending an interrupt signal to the process. A good program sometimes does some cleaning up before 'just dying'. An interrupt happens when you press control and c in a terminal where a process is running. The kill command sends signal 15 to a specified process id (kill 1000). If you want to send a control and c to the backgrounded process, use kill -2. If this fails, you can try sending a SIGKILL to the process (kill -9). The SIGKILL is very aggressive, as it tries to guarantee certain death by preventing the process from running its normal tidy-up.

SIGHUP is often sent to cause the program to reload its configuration files. (kill -1)

nohup command stops signal 1 getting into the process (so that closing a terminal shouldn't kill the command).

Introduction to Inter Process Communication (IPC)

- PIPES FIFOs
- Network socket
- Shared Memory
- RPC
- Unix Sockets

- In summary .. lots of ways

Systems Administrators and Developers work together to build an environment and software tool which can perform tasks that integrate different systems.

There are many ways to send information between different processes, sometimes it is not possible, or very inefficient to use one method rather than another.

- Are the processes running on separate machines ? If not today, how do we scale the application so that it can ?
- Do we need to stream constant data, or a snapshot of 'now' periodically, or do we need to leave messages for other software. Do we need to guarantee that the message gets through ?
- Are we running the same software on lots of machines in order to cluster the application ?

When we know the answers to the questions, a technology will seem obvious

- Pipes or named pipes.
- TCP/IP network connection and unique protocol
- Database
- Shared memory segments
- RPC

Files and Processes together - /proc

- Not every unix has a /proc. Even the unices that do, have a different kernel and therefore treats /proc differently.
- All processes have entries in /proc (every process is a file)
- Many kernel variables can be read or changed.
- Constantly changing
- Don't use editors to read or write to the stuff in /proc - see the examples in the notes.

Do not ever use an editor like vi on these files. Query the /proc data using cat, and write to it using echo.

```
root@blob:~# cat /proc/meminfo
      total:    used:    free: shared: buffers:  cached:
Mem:  128933888 104280064 24653824      0 45727744 22855680
Swap: 1085726720      0 1085726720
```

```
root@blob:~# echo 1 > /proc/sys/net/ipv4/ip_autoconfig
```

/proc/[process-id] is a directory that contains information about the running process which can be read with cat. (e.g. /proc/1000/cwd shows the directory that process id is running in)

Files and devices together - /dev

- We have covered securing access to a device through file security techniques

```
root@volpudl:~# ls -l /dev/lcdn3 /dev/sdc5
crw-rw---- 1 root tty 45, 3 1998-05-07 11:48 /dev/lcdn3
brw-r----- 1 root disk 8, 37 1995-04-29 11:34 /dev/sdc5
```

- b as the file type means 'block device' or 'This device can be a disk'
c as the file type means 'character device' - all other devices.
- Two two numbers separated by a comma are the major and minor numbers.
The major number specifies the driver number in an array of drivers. The
minor number is a device on that driver.
- Create device files with `mknod <name> <b|c> <major> <minor>`
`mknod` is part of GNU coreutils on Linux.

Block devices are any devices which can hold a file system - not just disks as we know them, but raid devices, memory cards, loopback devices, etc.

Character devices are 'everything else' - modems, serial ports, mice, video cards, etc.

`devices.txt` in the Linux source documentation is the definitive list of hardware types which can be created via `mknod`, ordered by major device number.

- We now know:
 - How unix file security works
 - Files are more than just 'documents' and 'binaries'
 - How this relates to the security of devices
 - How we can secure a program's operating environment

 - What a process is
 - The Parent-Child relationship of processes
 - How processes are scheduled
 - Job Control
 - What signals are & how to kill processes

- We also thought a little about IPC, /proc and /dev.

We now know quite a lot about how a unix-like operating system works. We've mastered handling files and processes, two of the most fundamental features of a 'unixy' OS.

Understanding everything we have learned today will help Sys Admins make good choices, but we now need to build upon the theory by looking at some of the tools at our disposal.

Managing Users

We have looked at how we can manage who (system and human users) can manage what (using file security, to secure access to devices, processes, and filesystem resources.)

User accounts

- On most unices, a user account is simply a valid entry in the `/etc/passwd` and `/etc/shadow` files.
- Creating a user account involves
 - Selecting a unique/distinct ID number
 - Selecting a group
 - Creating a `/etc/passwd` line
 - Creating the login directory and configuring it as the home dir in `passwd`
 - Changing the ownership rights of the home directory
 - Setting an initial password with `passwd`
- Luckily we have tools to do this ! On Debian Linux look at `adduser` - some unices have `'useradd'`. Ensure all of these tasks above are complete the first time after using a tool to manage users !

If the tools exist to create user accounts, use them.

Tools do not tend to be portable between unix versions, or even linux distributions - find out how to handle users properly with your own distribution.

Avoid editing `/etc/passwd` where possible with a text editor. Most unices and Linux distributions ship with a tool for safely editing this file - e.g. `chsh` to change the login shell, `chfn` to change other typically edited entries in the `/etc/passwd` line.

Note how the `passwd` file is just plain text, the ever-recommended storage mechanism !

Password Security

- It goes without saying that all passwords should be secure.
- Don't let system/daemon user accounts be logged into at all.
Lock accounts with `passwd -l` (account remains valid, just can't log in)
- Mixed case, alphanumeric, long passwords are the strongest.
- `pwgen` is a command which makes quite good passwords

```
andy@joy:~/backups$ pwgen -B 16
Nee9sheiwahgiug3 tooKahqu3osievoo oofuaah4aighe7ho oon4johqueueJ4jie
EcheemeiC9queeyu ocochec3au7doa4n Equah3pho4aikeeh u9eeng9jaine9eiF
thooj9vue9Hee7ow eo3AeNgho7eish3u oorah7aeg4aecii9 iegei9ahwitohiek
aixae7e3hoo4igie ahthooweba3Tiech eep7ye3ee9oowohk sheehu7Eikixahoo
```

```
.
.
.
```

Environment defaults

- /etc/profile sets default environment properties for the system.
- ~/.profile or ~/.bash_profile sets user specific environment properties

e.g. if a user wants to set their NNTPSERVER environment variable to news.individual.net, can do this themselves in their .bash_history profile.

if an admin wants to set the NNTPSERVER environment variable to news.bigcompany.com, they can default this in /etc/profile.
- /etc/skel is a default home directory that is copied over by most user-creation tools to set up a default 'rich' environment (company specific files or links, sample config files for various programs in use.)

An environment variable for a single shell session can be set with the 'export' command

```
export NNTPSERVER=news.individual.net
```

```
.  
.
.
```

no need to use export to set the environment variable for a single line... just set the variable and run the command.

```
NNTPSERVER=news.individual.net slrn
```

Removing Users

- Removing users is not just a case of locking their login, or home directory.
- If we agree that the definition of a user is a valid line in `/etc/passwd`, then this is what needs to be removed, to remove the user. Additionally, disk space can be saved, and process security can be ensured by removing the users home directory and files.
- We already know that editing the `/etc/passwd` file with a standard editor is bad, so we will want to find the tool to remove users on our unix - `userdel` on many typical linux distributions.
`userdel -r` to remove users home directory and files too.
If integrating into a script check the exit value and meaning in `man userdel`.



inetd

inetd is the internet superserver. It is a program which maintains 'passive sockets' on a variety of ports, so that when a connection from a remote computer is made, it can be passed to the correct application, without the application needing to manage its own attachments to a port.

When to and not to use inetd

- One program handing inbound connections, for many services.
- inetd only runs server programs as they are needed - lower overhead
- inetd not good for services accessed lots, and regularly - higher overhead
- inetd good when you write your own server software - much less coding.
- inetd lets you use tcpwrappers

inetd is a great tool for saving time. It works best when a network connection which will last a long time is needed, and connections will be relatively infrequent (e.g. telnet). It works least well (but still works of course) when the network service is a busy web-server or mail-server.

If you want to develop your own server software to do a very simple job (e.g. stream the status of an event to a remote collection) then you can use inetd to handle inbound connections. inetd sends inbound traffic to the STDIN of your script/program, and the STDOUT of your script is passed out to the remote user via inetd. Do not underestimate just how useful this is - it lets you write fairly complex network service software using nothing more complicated than a bash script.

tcpwrappers are access controls based on remote hosts which can be used to filter connections to services. Not all network services honour tcpwrappers, but inetd does. You can therefore write simple firewall policies (based only on connecting hosts) using the hosts.allow and hosts.deny files. hosts.deny takes priority - a host mentioned in hosts.deny and hosts.allow will be denied. Changing these files does not require a network restart - it's a runtime option.

How to configure inetd

- /etc/services - lists 'well known ports' of protocols

```
ftp-data 20/tcp
ftp      21/tcp
fsp      21/udp      fspd
ssh      22/tcp      # SSH Remote Login Protocol
telnet   23/tcp
smtp     25/tcp      mail
tftp     37/tcp      tftpsrvr
time     37/udp      timesrvr
rtp      39/udp      resource    # resource location
nameserv 42/tcp      name        # IEN 110
whois    43/tcp      nicname
tacacs   49/tcp      # Login Host Protocol (TACACS)
tacacs   49/udp
```

- /etc/inetd.conf - lists which ports inetd will listen to

```
#discard stream tcp nowait root internal
#discard dgram udp wait root internal
#dshime stream tcp nowait root internal
#lpc stream tcp nowait root internal
telnet stream tcp nowait telnetd /usr/sbin/tcpd /usr/sbin/in.telnetd
ident stream tcp wait identd /usr/sbin/identd identd
```

- On Debian, the 'harden-servers' package blocks non encrypted inetd defaults

The /etc/services file details a lot of well known port numbers and service names. This is a handy file to have when looking up what may be (not what is definitely) running on a port....

The /etc/inetd.conf file details a list of ports (either by a mapping from a service name to a port in /etc/services, or an explicit port number) to applications which run. Some of the services are inetd specific/built in, but some call external pieces of software.

When a connection comes in on port 23, inetd.conf knows that port 23 is mapped to the 'telnet' service via services(5), and secondly, knows that it should start the in.telnetd program thanks to the configuration line in inetd.conf(5).

Writing “My First Server”

- A simple script which outputs the kernel version running on a Linux server

```
#!/bin/bash
KERNEL=$(uname -r)
HOSTNAME=$(hostname)
echo "$HOSTNAME is running kernel version $KERNEL"
```

- We want to be able to query this remotely so that a sys-admin can check kernel version without logging in to the server.

```
1000 stream tcp nowait admin /usr/local/bin/whichkernel.sh
```

- Restart inetd... then try to connect

```
factory:~ and$ telnet -N 10.33.3.51 1000
Trying 10.33.3.51...
Connected to 10.33.3.51.
Escape character is '^['.
trollied is running kernel version 2.6.18.3
connection closed by foreign host.
```

- You have just written a server in four lines of bash.

The inetd configuration file options are specified in ‘man 5 inetd.conf), but the important parts here are ‘1000’ - the port to listen out on, ‘admin’ is the user to run the script as, and the name of the program to run.

As a general rule, you must pay close attention when exposing software to the internet, and try to avoid writing software packages that don’t require authentication if private data might be revealed. When authentication and encryption are desirable, you could write a script which is set as a user’s ‘shell’, and then force users to login with a shared username and password. This buys a little security (login required & encryption) - although it is possible to wrap telnet in ssl so that telnet sessions are encrypted too.

xinetd

- More feature rich replacement, defaults on some distributions, e.g. Red Hat.
- Each service is a separate file, easier to maintain packages
- Many more features, including only allowing connections to a port at various specific times of day.
- Can place restrictions on numbers of servers that the system can start in a given time frame
- Configs for services go into /etc/xinetd.d

Example xinetd.conf service file.

```
service telnet
{
    disable = no
    flags      = REUSE
    socket_type = stream
    instances = 5
    nice = 10
    access_times = 9:00-17:00
    wait      = no
    user      = root
    server    = /usr/sbin/in.telnetd
    log_on_failure += USERID
}
```

Demonstrates some advanced features of xinetd - this service is only exposed during the 9-5 working day, and runs with a relatively low priority. Defaults go into /etc/xinetd.conf

Bash Scripting

Bash scripting is an important part of a systems administrator's job, and can be used for a variety of tasks

- monitoring
- emailing reports
- simplifying jobs
- daily batch jobs
- ensuring common tasks are completed in a homogenous way

It is excellent practice to script regularly requested tasks. When a script is mature, it can be incorporated into an application that empowers users with managing their own details.

The anatomy of a bash script.

- A simple script to cleanly set the hostname

```
#!/bin/sh
PATH=/sbin:/bin
. /lib/init/vars.sh
. /lib/lib/init-functions

[ -f /etc/hostname ] && HOSTNAME=$(cat /etc/hostname)
# Keep current name if /etc/hostname is missing.
[ -z "$HOSTNAME" ] && HOSTNAME=$(hostname)
# And set it to 'localhost' if no setting was found
[ -z "$HOSTNAME" ] && HOSTNAME=localhost

hostname "$HOSTNAME"
ES=$?
[ "$VERBOSE" != no ] && log_action_end_msg $ES
exit $ES
```

- This script demonstrates the
 - shebang line
 - use of variables
 - clean handling of exit codes
 - conditionals
 - importing variables and functions from other files.
 - comments
- We will look at these features in detail.

This script is actually a 'quite modified' version of the script which is used to set the hostname on a debian server. Essentially a bash script is a structured file that contains a list of commands to run, that would make sense on the bash command line.

All scripts should start with a line detailing which interpreter to use to handle the script. This line is called the 'hashbang' or 'shebang'. The shebang characters (#!) equal the byte string (0x23 0x21) which denote to the OS that this file is an executable script which needs an interpreter. Some OSes still use this magic number for identification, so in the interests of portability you should consider this line to be essential. It is best practice to add a space after the shebang characters as some OSes treat the magic number to be "#!/" as the first four bytes of the file.

Any interpreter can follow a shebang, e.g.

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

You can incorporate another bash script into yours with the 'source' command or just a dot as shown in the example above. This is useful if you want to share environment variables or bash functions which are common to a family of scripts, and is much better practice than copy and pasting all the code a lot of times into a lot of scripts ! It means you have just one place to change if you want to change a shared feature.

Shell variables

- Essentially similar to environment variables, only apply to script lifetime.
- Not clever (no associative arrays!)
- Typically use string (text) and integer (whole number) arrays in bash.
- Create variables :
VARNAME="variable contents"
No spaces around the = are allowed.
use 'declare' builtin command to process variables safely
- Variable names are case sensitive, and normally expressed in capital letters.
They must not start with a number.

Variables are quite handy -

- define something once that you want to use in a homogenous manner in a script (e.g. location of a program)
- make a config block at the top of your script which is easy to maintain.
- can store output of commands in them and use the output in other places in your script.

4 types of variables - string (anything), integer (numeric), array (list), constant (read-only).

'declare' options are -a (force variable to be created as an array), -i (force integer), -r (read only). e.g. declare -i VAR=1

Dealing with numbers in variables.

- Wrap arithmetic expressions in `$(())` notation

```
#! /usr/bin/bash
declare -i ONE=1
declare -i TWO=2
declare -i THREE=3
echo=$((ONE + TWO))
echo=$((TWO * THREE))
factory:- andy$ sh numbers.sh
3
6
```

- Get the length of a string variable as `${#VARIABLE}`

Lots of useful number based operations, including `VARIABLE++` which can be used as a counting aid.

Special variables

\$0	Expands to the name of the shell or shell script
\$\$	Expands to the process ID of the shell
\$?	Expands to the exit status of the most recently executed foreground command
\$!	Expands to the exit status of the most recently executed background command
\$#	Expands to the number of positional parameters (options after the name of the script, e.g. ./script.sh opt1 opt2 opt3)
\$1..	Expands to the positional parameter (e.g. \$1 would expand to 'opt1' in example above, \$2 is 'opt2')

These are important built-in features of the bash shell.

For example, it may be very important for you to catch the exit code of a program which you run from your script, and complain loudly where this did not execute properly. This is often therefore just a case of ensuring that \$? is 0 ..

The positional parameters are normally used to define options in your script, e.g.

```
./script admin@company.com
```

causes the email address 'admin@company.com' to be in the \$1 variable - this can be used at various points in the script as an email address to contact someone on in the event of a failure, and is cleaner than hardcoding addresses all over your script.

Fragmenting variables

- `LONGVAR="abcdefghijklmnopqrstuvwxy"`
- `echo ${LONGVAR:5}`
`fg hijklmnopqrstuvwxy`
- `echo ${LONGVAR:2:3}`
`cde`
- Syntax is `${VARIABLE:offset:length}`
- Don't use for columns! Use `awk`.
`cat tabbed-data.txt | awk '{print $3}'`
prints third column (separated by spaces) in the file 'tabbed-data.txt'.

A 'poor man's cut'.

Only works if you know that variable length is always the same, and the data you care about will be in the same place.

Not very good at sorting column data. e.g. to get a list of running process ids from the `ps` command use
`ps -aux | awk '{print $1}'`

Repetition with for

- Used to loop through a list of values
- for NAME [in LIST]; do THINGS; done

```
andy@grumps:~/bashtest$ ls
file1 file2 file3
andy@grumps:~/bashtest$ for FILE in `ls`; do cp "$FILE" "$FILE.backup"; done
andy@grumps:~/bashtest$ ls
file1 file1.backup file2 file2.backup file3 file3.backup
```

- NAME is a bash variable that contains a single value.
- The script loops around until all possible values expanded by the 'in' rule are evaluated by the block of code between 'do' and 'done'.

for is a very useful loop in bash, and builds upon what you know about variables. The worked example on the slide shows how it can be used to create backup files.

the 'in' section of the for command does not need to be another command, as it is in this example, it could be a defined list :

```
andy@grumps:~/bashtest$ for NUM in 1 2 3 4; do echo $NUM; done
```

```
1
2
3
4
```

Repetition with while

- Used to loop, given a circumstance
- while CONTROL; do THINGS; done
- Typically, though not required, the commands in THINGS will change the command that is CONTROL to change, otherwise the loop may be endless
- You may want an endless loop (e.g. wrapper script.)
- What happens if the line which increments COUNT is before the echo statement?

```
andy@groups:~$ cat bashwhile.sh
#!/usr/bin/bash

COUNT=0
while [ $COUNT -lt 4 ]
do
  echo "Loop finished $COUNT times"
  COUNT=$((COUNT+1))
done
echo "Bored of this"
andy@groups:~$ bash bashwhile.sh
Loop finished 0 times
Loop finished 1 times
Loop finished 2 times
Loop finished 3 times
Bored of this
```

While evaluates an instruction to decide whether it is true, and if it is, will run a block a code. In the worked example in the slide, we evaluate whether a variable called COUNT contains a value less than four, and if it is, it runs the code in the block.

If nothing varies the COUNT variable, this is an endless loop. Can also simulate an endless loop with `while [1=1]` or `while true`

Can do clever things like evaluate the exit status of a command, and only run a while loop whilst the consequent commands inside the loop return 0 (or return an error!)

A third type of loop called the until loop has very similar syntax, but causes commands to run until the control command is not true. e.g. in the example above, `until [$COUNT -lt 4]` would cause the script not to run the block of code, as COUNT is 0 at the start (0 is less than 4). To do a similar task to the above, we would say `'until [$COUNT -gt 3]'`

Conditional Statements

- We touched briefly on a conditional - evaluating whether a variable contained an integer smaller than four in our while-loop. The way to handle conditionals in a script is to use the 'if' command.
- if TEST; then CONSEQUENT ; fi

```
andy@grumps:~$ cat ifuser.sh
#!/usr/bin/bash
# The user's ID is stored in a builtin
# variable called UID

if [ $UID -gt "99" ]
then
echo "You are a mere mortal with no super-powers!" ;
fi

if [ $UID -lt "100" ]
then
echo "I fear your powers, great one." ;
fi
andy@grumps:~$ bash ifuser.sh
You are a mere mortal with no super-powers!

root@grumps:~# bash /home/andy/ifuser.sh
I fear your powers, great one.
```

Clever conditional statements with files

- if [?? FILE] - replace ?? with

	True if
-a	FILE exists
-b	FILE exists and is a block device
-c	FILE exists and is a character device
-d	FILE exists and is a directory
-f	FILE exists and is a regular file
-h	FILE exists, and is a symbolic link
-r	FILE exists and can be read
-s	FILE exists and is larger than 0 bytes
-w	FILE exists and is writable
-x	FILE exists and is executable
-N	FILE exists and was modified since last read

Scripts should never return an error, so it's very handy to make sure a file exists before operating on it.

Sometimes you actually want your script to do little more than check a file exists and is being written to properly, e.g. monitoring a log file...

Conditionals for Champions

- If [! TEST] - if the test is not true
- if [TEST1 -a TEST2] - if test1 and test2 are true
- if [TEST2 -o TEST1] - if test2 or test1 are true

- else can be used so that an 'if' always does something
if [1=1] ;
then echo "one equals one" ;
else echo "one no longer equals one! ftw" ;
fi

Can also string together lots and lots of if commands using the elif statement

```
if TEST; then
  do something ;
elif DIFFERENT TEST; then
  do something else;
else
  do something really wild;
fi
```

But this is not particularly easy to read or maintain, so it's much more likely that a REAL champion of conditionals will use a 'case' statement.

Case statements (elif for champions)

- Nested if statements are useful, but can be confusing and hard to maintain.
- Most initscripts provide a good example of how 'case' works by evaluating the \$1 variable (command option) and reacting differently based on its contents.
- case \$MYVAR in THIS) do this;; THAT) do that;; *) do something else;; esac
- space your case statements out or they look like the above.

Pinched from the TLDP site

```
#!/bin/bash
# This script does a very simple test for checking disk space.
space=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 | cut -d "%" -f1 -`
case $space in
[1-6]*)
    Message="All is quiet."
    ;;
[7-8]*)
    Message="Start thinking about cleaning out some stuff. There's a partition that is $space % full."
    ;;
9[1-8])
    Message="Better hurry with that new disk... One partition is $space % full."
    ;;
esac
echo $Message | mail -s "disk report `date`" root
```

A quick introduction to functions

- Functions are a uniquely named group of commands. Specified with either :
function FUNCTIONNAME { commands; }
or
FUNCTIONNAME () { commands; }
- Handy if you want to run the same block of commands in a family of scripts (like shared libraries in more advanced programming languages), or want to use the same block of commands lots of times in one script.
- The body of a function should end with a semi-colon ;
- display functions in your running shell with the which command

```
andy@grumps:~$ cat spesh.sh
#!/usr/bin/bash

function HOWDULL
{
    TODAY=$(date '+ %A')
    echo "This is the dulllest function that you will ever see on a $TODAY";
}

HOWDULL

andy@grumps:~$ bash spesh.sh
This is the dulllest function that you will ever see on a Monday
```

Shell functions are very handy.. especially if you want to run the same code in many places. In terms of maintenance, having functions means you only need to update your code in one place.

the example on the slide isn't a perfect example as there's no real need to put the HOWDULL function into a function at all.. it's only called once, and that's immediately after the function is created ! However, this demonstrates the structure of a function.

Catching user input

- Use the read command. read VARNAME.

```
andy@grumps:~$ cat spesh.sh
#!/usr/bin/bash
echo "What is your name?"
read NAME
echo "Hello, $NAME"
andy@grumps:~$ bash spesh.sh
What is your name?
Andy
Hello, Andy
```

- read -n x -reads x characters from stdin
- read -e - use readline (completion, lookback)
- read -d x - use x as a delimiter, not a newline as a delimiter
- read -s - silent - don't echo characters to screen on a terminal.

Signals in bash scripts

- Signals were introduced to you earlier as a way for one process to send a message to another process (e.g. pressing ctrl+c at the terminal)
- Use the 'trap' command to trap a signal
- trap COMMAND SIGNAL, e.g. trap exit 1 SIGINT

```
factory:~ andy$ cat neverending.sh
#!/usr/bin/bash
trap "echo Ouch!!!; exit 1" SIGINT
while true
do
    echo "I love being a shell script"
    sleep 10
done
factory:~ andy$ bash neverending.sh
I love being a shell script
I love being a shell script
Ouch!!!
```

e.g. SIGALRM to handle time out.

```
#!/bin/bash
sleep 5 && kill -s 14 $$ &
trap timeup 14
```

```
timeup()
{
    echo "Time up .. abort!"
    exit 1
}
for i in 1 2 3 4 5 6 7;
do
    echo $i
    sleep 1
done
```