Linux and UNIX Basics

By JL@HisOwn.com

Jonathan Levin
Training & Consulting

V 1.01 – 5/23/2007

Table of Contents

About the Author:

Jonathan **L**evin specializes in training and consulting services. This, and many other training materials, are created and constantly updated to reflect the ever changing environment of the IT industry.

To report errata, or for more details, feel free to email **JL@HisOwn.com**

Introduction

What is UNIX

- Originally contrived in Bell Laboratories
- Designed as a single user O/S
- Re-written in C, 1975
- An O/S by programmers for programmers
- User friendly, but very picky about who its friends are…

UNIX started out as a single user operating system for the PDP 7/11 platforms, in Bell Labs. Its name was coined as a pun on Multics, which was a common OS.

UNIX has since evolved immensely: the first major breakthrough was introduced in 1975 – the OS was rewritten in C – allowing for portability between platforms. The original C source has since then been licensed by IBM, HP, SGI, and other vendors in their own interpretations of the OS. SCO currently holds the rights to the UNIX source (and was threatening to sue IBM for letting Linux in on it…).

Introduction

Operating System Structure

Two roads diverged in the UNIX wood:

BSD – Berkeley Software Development
AT&T – (now known as System V)

General experience is always roughly the same.

Implementations differ in API, kernel structure and some user commands (e.g. at, ps, lp/r).

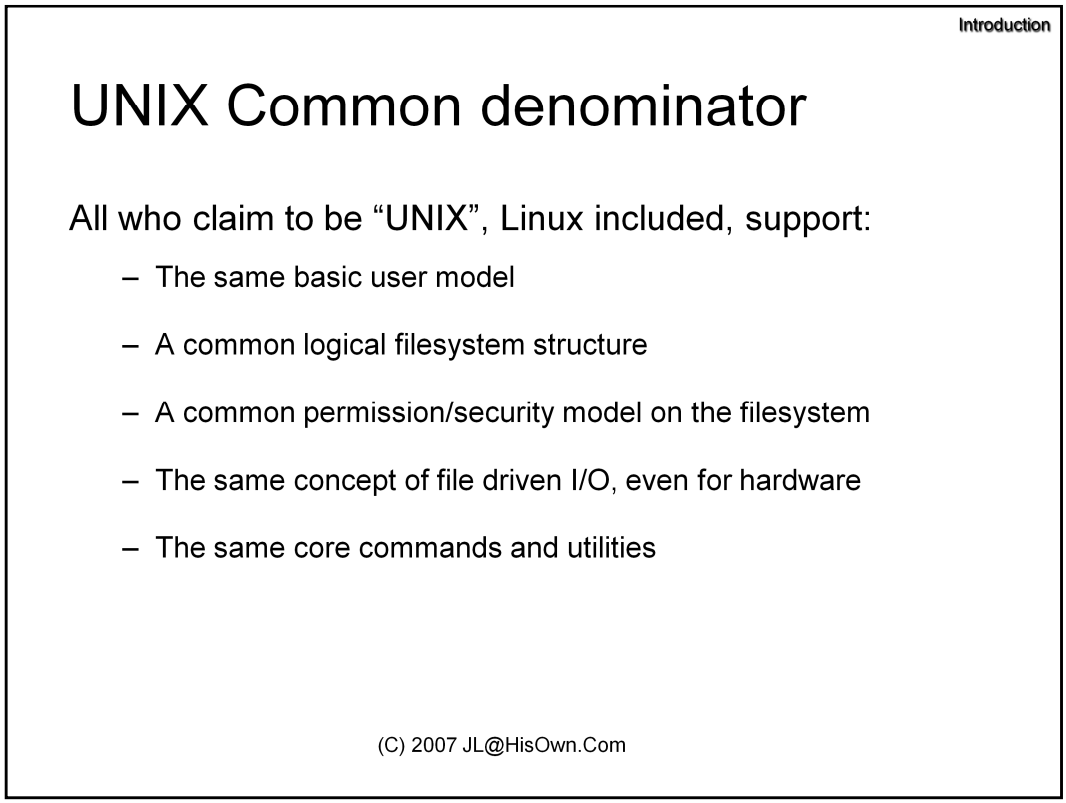
Almost all UNIX types support the POSIX standard.

There is no SINGLE UNIX. There are plenty of flavors, however.

Linux is the most common UNIX nowadays, owing to its open sourced and free nature (although (Free|Open|Net)BSD are also open, they are hardly as popular). The GNU utilities (also open source)

The following table shows a partial listing of UNIX flavors:

Vendor	Brand	Versions	Processor	Standard
Sun	SunOS (Solaris)	2.5,2.6,7,8,9,10	Sparc/x86	System V
HP	HP-UX	10.20, 11, 11.11, 11.11i, 11.23	PA-RISC	System V + POSIX
Digital	Keeps changing..	3 (Ultrix) 4 (Digital UNIX) 5 Tru64	Alpha	OSF + SysV
SGI	IRIX	5.3, 6.2,6.4,6.5.x	MIPS	OSF + Sys V
IBM	AIX	4.0-4.3, 5.0.. 5.3	RS6000 +	BSD
Linux	RedHat, SUSE..	Kernels 2.4, 2.6	Any	Hybrid/GNU



Despite the multiple flavors of UNIX, they all have a rather large common denominator:

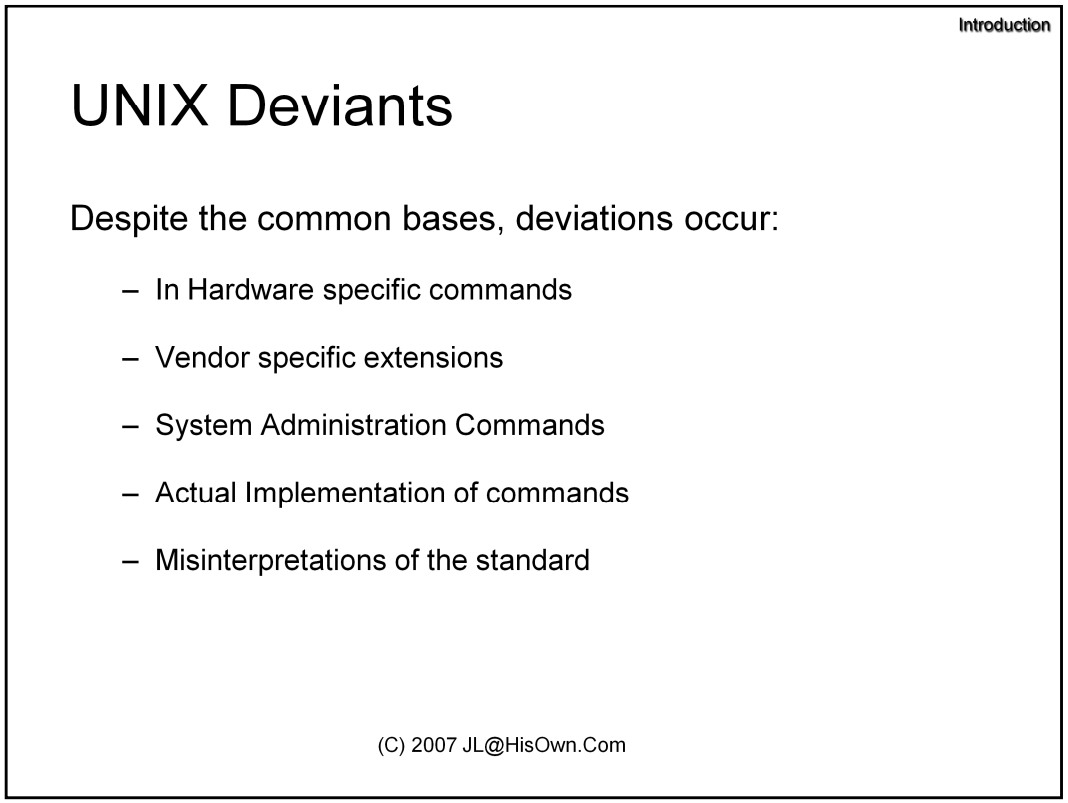
The same basic user model – out of box, all UNIX types have the same administrator usernames, such as "root", "bin", "adm", "lp". The user "database" is identical or similar in all.

A common logical filesystem structure – All UNIX flavors use the same logical filesystem structure, with the same directory names and functions – like "bin", "sbin", "lib", "etc", "tmp".. File handling commands, like "cp", "mv", "ls", are also common.

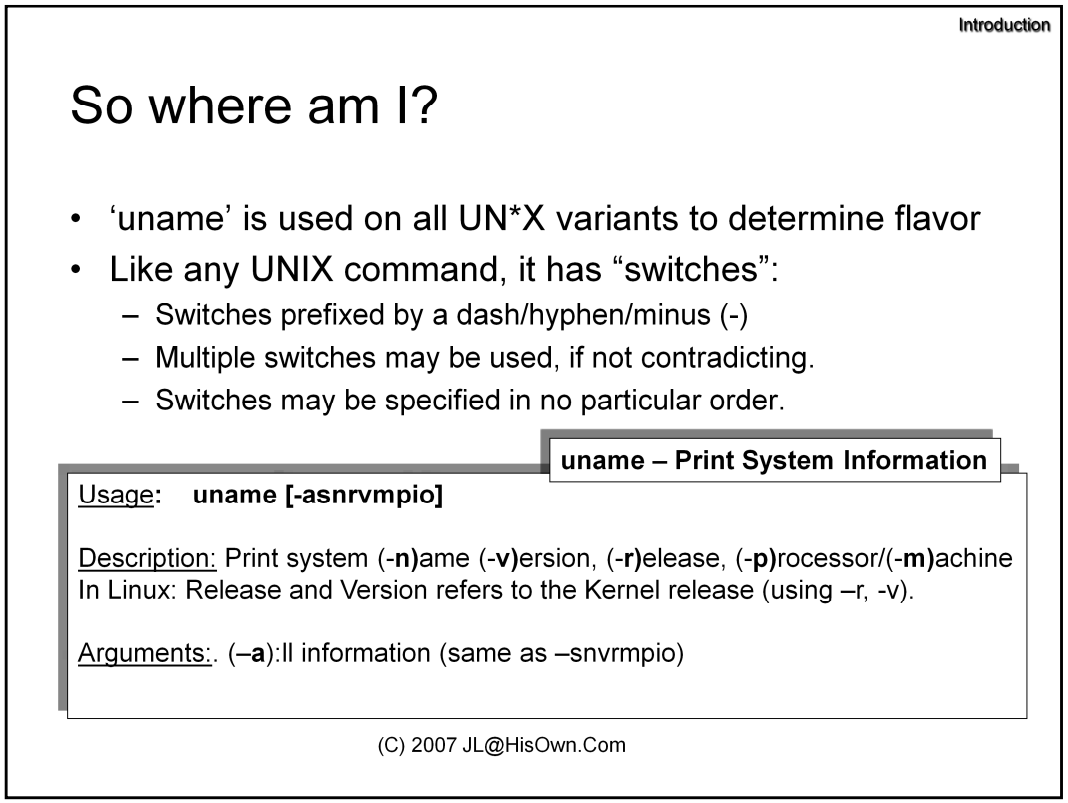
A common permission/security model on the filesystem – The file-security commands are also similar – and are referred to as the "holy trinity" of chmod/chown/chgrp. Although this basic model has been extended in several UNIX types, as it is extremely limited.

The same concept of file driven I/O, even for hardware – In UNIX, ***everything*** is a file. Even hardware devices can be opened and accessed just like files.

The same core commands and utilities – as stated above, file handling and security commands are identical or similar. Process handling commands, and the entire CLI is common enough for one UNIX denizen to migrate to a new environment without noticing any impediment.



However, UNIX flavors still differ substantially. Fortunately, it's the advanced functionality, such as system-administration or hardware management commands that get very specific. So we're not likely to see too many of these differences here.



With all the variants of UNIX out there, the best way to find out what system you're on is the "**uname**" command – and this will therefore be the first command presented here.

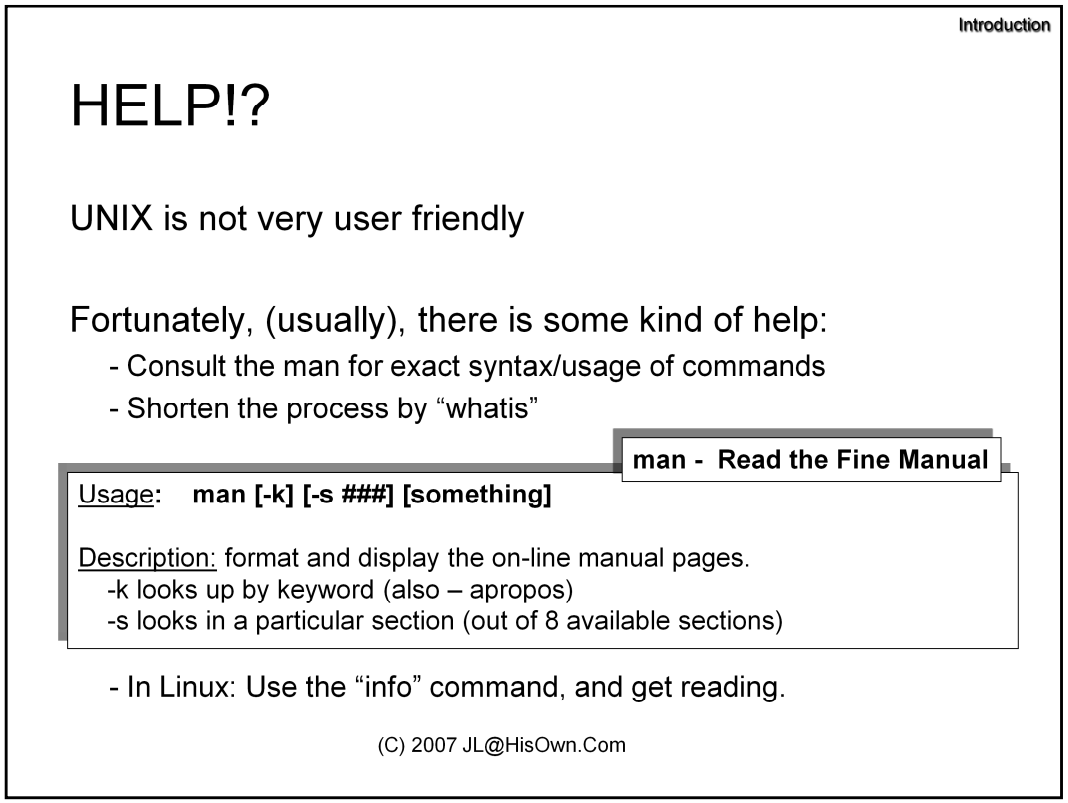
As a typical UNIX command, we can use uname to demonstrate the use of switches. Notice this output from a Linux system, where the switches were employed one by one:

```
jormungandr (~) $ uname -n
jormungandr
jormungandr (~) $ uname -s
Linux
jormungandr (~) $ uname -m
i686
jormungandr (~) $ uname -i
i386
jormungandr (~) $ uname -p
i686
jormungandr (~) $ uname -o
GNU/Linux
jormungandr (~) $ uname -v
#1 SMP Tue May 2 19:32:10 EDT 2006
jormungandr (~) $ uname -r
2.6.16-1.2107_FC4smp
jormungandr (~) $ uname -a
Linux jormungandr 2.6.16-1.2107_FC4smp #1 SMP Tue May 2 19:32:10 EDT 2006 i686
i686 i386 GNU/Linux
jormungandr (~) $ uname -nsmpoivr
Linux jormungandr 2.6.16-1.2107_FC4smp #1 SMP Tue May 2 19:32:10 EDT 2006 i686
i686 i386 GNU/Linux
```

Uname exists on all UNIX platforms, regardless of race, color or creed. Its output, however, is different, and corresponds to the system in question. On Solaris, the same command would get different results.

```
Surtr (~) $ uname -a
uname -a SunOS sng 5.8 Generic_108528-21 sun4u sparc SUNW,Ultra-Enterprise-10000
Surtr (~) $ uname -X
System = SunOS
Node = Surtr
Release = 5.8
KernelID = Generic_108528-21
Machine = sun4u
BusType =
Serial =
Users =
OEM# = 0
Origin# = 1
NumCPU = 4
```

Notice the output is different, and Solaris also supports a different switch (-X) that Linux, for example, does not. But the general gist of the command is pretty much the same.



UNIX is, to say the least, not very user friendly. Although most avid UNIX-philes will tell you it is – but it's picky as to who its friends are.

The UNIX idea of "help" is in the "man" command – or, in UNIXese – RTFM. The entire manual (a hefty tome) is available (in text form, of course), and "man" enables you to view whatever command is of interest.

UNIX Manuals are divided into 8 sections:

Section	Description
1	User commands – Most of what you want is here
2	System calls – For lower-level developers only
3	Programming Interfaces - Most of what developers want is here
4	File Formats / Device Drivers / Configuration (varies)
5	File Formats (Configuration Files)
6	Games (Yep. You read right. But usually this section is empty..)
7	Device Drivers (don't go here)
8 (or 1(M))	System Administration commands

The UNIX CLI

Working with shells

UNIX CLI

The UNIX (primitive) User Interface

Most user interaction is performed over *terminals*

- physical (ttys): Are actual connected devices
- pseudo (pts) – e.g. telnet, ssh, X – are network emulated

tty– print the file name of the terminal connected to standard input

Usage: **tty**

Description: Print system (-**n)**ame (-**v)**ersion, (-**r)**elease, (-**p)**rocessor/(-**m)**achine
In Linux: Release and Version refers to the Kernel release (using –r, -v).

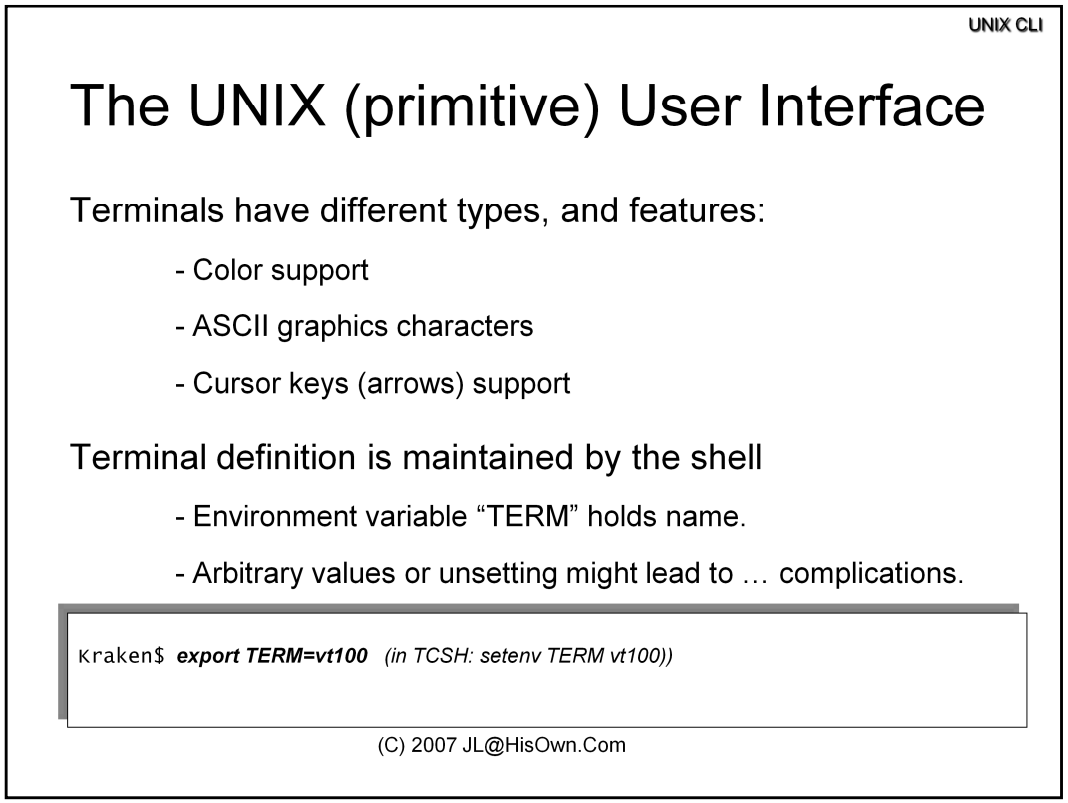
Arguments:. (–**s**):ilent = no output (useful only in scripts)

Terminal session maintained by a command *shell*.

- Many varieties exist: sh, ksh, bash, csh, tcsh, zsh…
- Shells allow the execution of commands, much like in DOS.

The command environment you are in, when you type in the various commands, is called a ***shell***. This is a command interpreter, not unlike any MS-DOS command prompt – but FAR stronger in features, as we will see later. Each such "shell" usually runs in the context of a "terminal" – a relic from the olden days when UNIX had many dumb terminals connected to one large server machine.

To see the "terminals" in action, one uses the "tty" command. This simple command merely returns the terminal identifier. Terminals are generally classified as physical (/dev/tty..), or virtual, or pseudo (/dev/pts/..).

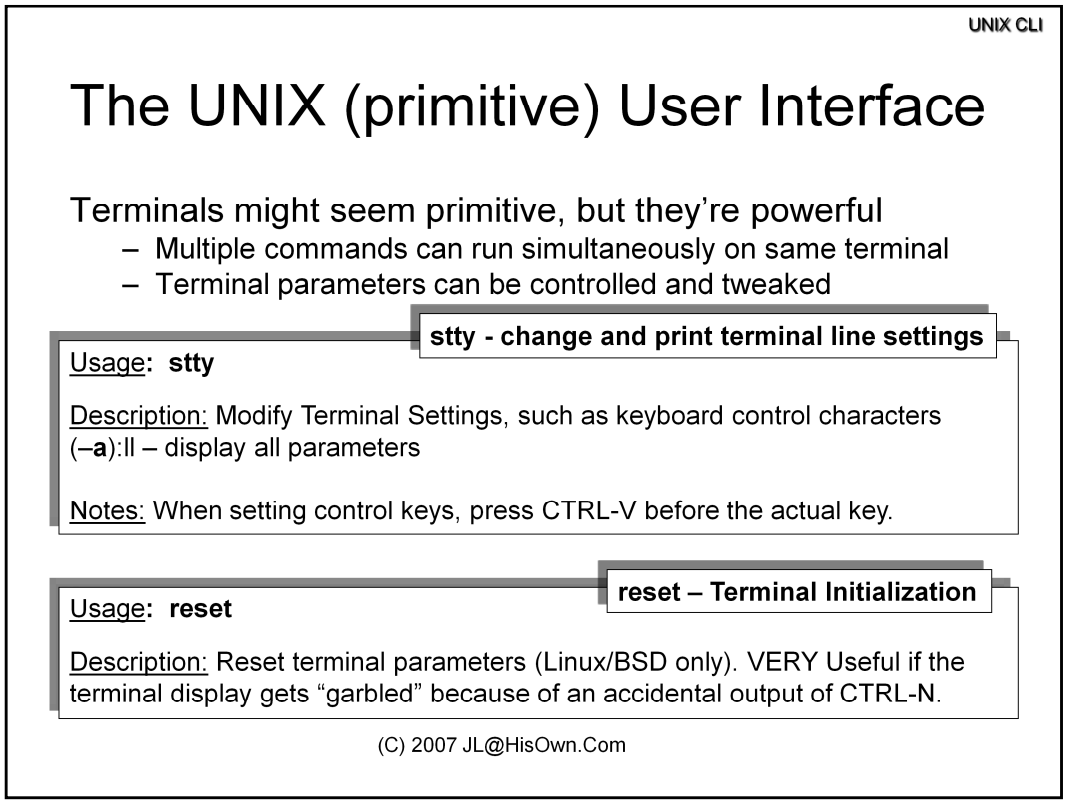


Terminals also have particular "capabilities" – such as colors, cursor keys, and other features. Not all terminals share the same features.

UNIX maintains a "Terminal Capabilities" database, in a file (called /etc/termcap) as well as specific terminal entries in a directory (/usr/share/terminfo). These enable the various UNIX commands to remain agnostic to the type of Terminal used, as the system "translates" capabilties such as full screen, color codes, etc without the need to code them specifically into each and every terminal.

As the following example shows, setting the terminal makes a difference for commands such as "vi" – the visual editor, that require "advanced" capabilities such as full screen.

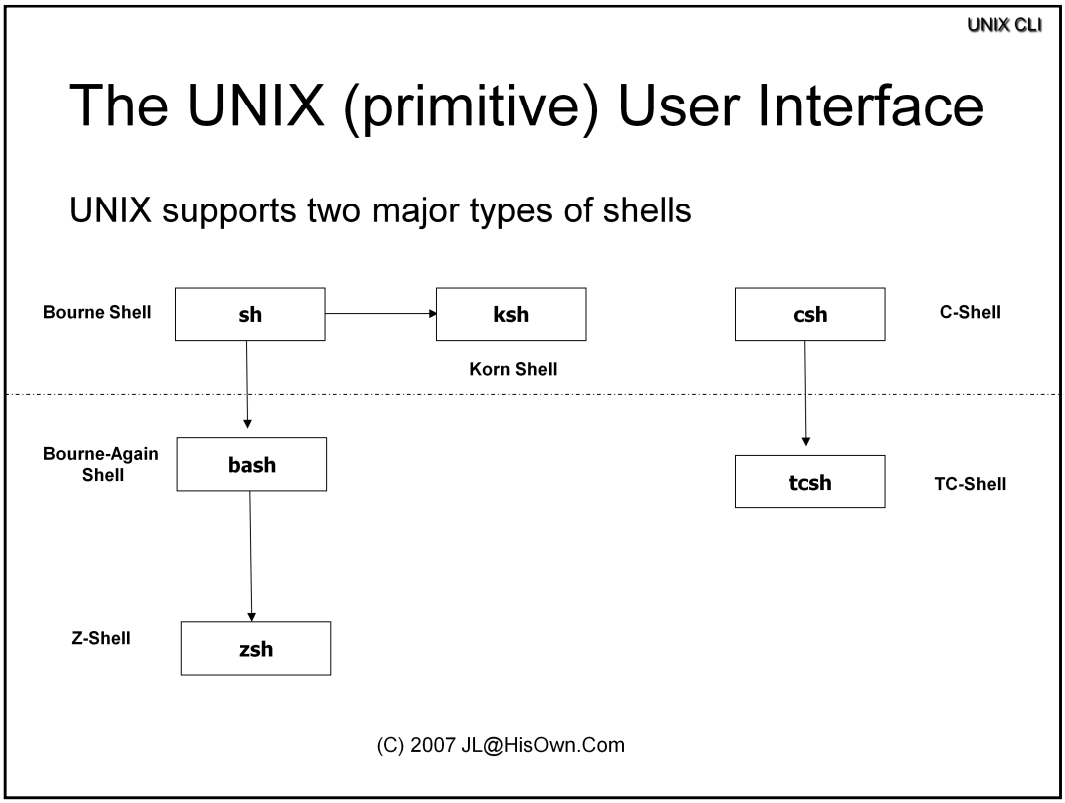
```
Kraken$ export TERM=xxx
zsh: can't find terminal definition for xx
Kraken$ vi
E558: Terminal entry not found in terminfo
'xx' not known. Available builtin terminals are:
   builtin_ansi
   builtin_xterm
   builtin_iris-ansi
   builtin_dumb
defaulting to 'ansi'
```



The 'stty' command is one of the least understood and less used commands in the UNIX environment – but sometimes it can be a lifesaver.

```
Kraken$ stty –a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke

Kraken$ stty erase ^H
```



The Shell is the single most important command in UNIX, since it maintains the user session and environment, and launches other commands. However, there are numerous types of shells, and your local UNIX or Linux might vary.

All UNIX flavors come with three installed shells: sh, ksh, and csh. Non-Standard shells can be found that extend those three, and all fall into the two basic categories shown on the slides: The one started by “sh” (The Bourne Shell), and the one started by “csh” (The C-Shell).

The Bourne shell, named after its inventor, is the most basic and limited of all the shells. Chances you’ll encounter it directly are fairly few, since it has been superseded by the open-source yet now almost standard BASH, or “Bourne Again” shell (a clever pun on the name of the original). Bourne shell itself is extremely restricted and with very little functionality, but it is still used as a common denominator for scripts. BASH, however is very common, and has been standardized as the default shell of choice in Linux, as well as newer versions of Solaris.

The Korn shell is a stronger version of Bourne, with enhanced syntax that is especially suited for scripts. It’s still a VERY inhospitable shell to be in, yet it does have significant advantages, even over the newer shells, when it comes to command or “job” control, as we will see later.

Zsh is the author’s favorite. Not always installed by default, it still is part of any Linux (or later Solaris) distribution, with very nifty enhancements not available anywhere else.

Csh, and its successor, tcsh, are named after the syntax they use – like the programming language of C, which is totally incompatible with those of the Bourne/Korn variety. The syntax differences will be demonstrated as we go along. One difference (‘export’ in sh/ksh vs. “setenv” in c/tc) was already shown.

The UNIX (primitive) User Interface

Shells are highly customizable by means of:

- Setting special environment variables
  - In sh/ksh: use “export”
  - in t/csh: use “setenv”

Common env. variables: TERM, PATH, HOME (and see below)

- Setting command aliases
  - In sh/ksh: use “alias …=…”
  - In t/csh: use “alias … …”

Command aliases are a very simple yet powerful mechanism to customize the shell. They enable to create your own custom commands, which are merely syntactic sugar for the actual commands in question. The syntax in all shells is roughly the same, with the main difference being the use (or lack) of an equals (=) sign. The alias usually replaces the command, and not the arguments. TCSH, however, supports aliases which are more powerful, enabling alias substitution as well.

We already saw that TERM environment variable. But it’s not the only one. We have several important variables maintained by the shell:

Variable	Use
HOME	Set to the home directory. In ‘cd’ callable as ~, or with no arguments
PATH	In sh/ksh/bash: $PATH, with : separators between directories In c/tcsh: $path (as an array, in parentheses with spaces separating
LD_LIBRARY_PATH	Same as PATH, but for library loading (wicked useful, but advanced)
TERM	Controls Terminal ID for full screen/color-enabled applications
SHELL	Semi standard – sometimes $shell, instead. Not to be relied on
PROMPT	Your shell command prompt. In tcsh – prompt. In KSH – PS1
OLDPWD	The previous working directory. In ‘cd’ also callable as “-”
PWD	Automatically set to what the present working directory is
USER	MAY be set by your login script to your username
WATCH	Zsh specific: allows you to watch for other users’ logins to system

UNIX CLI

The UNIX (primitive) User Interface

Bash, Zsh and tcsh allow for filename completion:

Bash: TAB completes or shows options

Tcsh: TAB completes, ^D shows option

- set autolist to show completion options on TAB
- set fignore to ignore file suffixes on completion

Zsh: Programmable command/argument completion

- Can complete command line ARGUMENTS!
- “man zshcompsys” for more options

Filename completion is the same idea as you have in DOS (using DOSKEY) or NT/2000 (using CMD’s CompletionChar registry key). Zsh takes it to an entirely new level by programmatically enabling the completion of command line switches (which are command specific) and not just arguments (as filenames)

UNIX CLI

The UNIX (primitive) User Interface

Shells special characters:

Character	Meaning
!	History (bash, tcsh zsh)
&	Run command in the background
>,>>,<	Output/Input Redirection
|	Piping (Redirection between commands)
$	Variable recall
"	Quoting, Variable recall allowed
'	Quoting, verbatim
( )	Command Execution
? *	Wildcards (? = single, * = any)

The above characters are common to all, or most shells.

The following example shows some of the "quoting". Background and history are shown in the exercise.

```
Kraken$  echo $TERM
linux
Kraken$ echo "$TERM"
linux
Kraken$ echo '$TERM'
TERM
Kraken$ pwd ; (cd /etc ; pwd) ; pwd
/
/etc
/
```

The UNIX (primitive) User Interface

TCSH also allows for command or argument correction:

```
Kraken (tcsh)$ set correct=cmd
Kraken (tcsh)$ lf
CORRECT>ls (y|n|e|a)?

Kraken (tcsh)$ set correct=all
Kraken (tcsh)$ moree /etc/passwdd
CORRECT>more /etc/passwd (y|n|e|a)?
```

Zsh support more of the same by “setopt”

```
Kraken (zsh)$ setopt correct
Kraken (zsh)$ lf
You meant ls, right?
Kraken (zsh)$ setopt correctall
Kraken (zsh)$ lf /etc/passwdd
You meant ls, right?
You meant /etc/passwd, right?
```


Command correction is VERY useful, especially in your early days of the UNIX experience. It’s one good reason to forsake bash in favor of the better, but non-standard zsh. It’s not perfect, but zsh has some pretty smart algorithms for deducing the right command for the right time.

The UNIX Command line environment

1. Getting to know the shell differences

If available, go into ksh, by simply typing “ksh”. Type a few commands, then press the UP arrow key, as if to recall a previous command. What happens?

_____

To fix this, type “set –o vi”. Now use the arrow keys (or press ESCAPE and use ‘j/k’ for up/down, and ‘h/l’ for left/right) to recall previous commands. To edit a recalled command line, use ‘a’, ‘x’, ‘i’ and ‘r’. What is the effect of pressing these keys?

_____

_____

Use the “pwd” command to see what directory you are currently in. Then try “echo $PWD”. Both commands show the same output. What is the difference between them?

_____

_____

Next, set the prompt. Type the following string: **`export PS1="$PWD >"`** (note: double quotes) what happens. What is this useful for?

_____

Next, ‘cd’ to a different directory. You’ll note the prompt hasn’t changed, and is “stuck” in the previous directory. Why?

_____

_____

Repeat the prompt setting, this type using single quotes instead of double quotes. What is the effect you see? Why?

_____

Repeat the above procedure for “tcsh” and “zsh”. What are the differences you see?

The UNIX Command line environment – Cont.

2. Job Control

This exercise demonstrates the versatility of command control, called “jobs” in the various Shells.

To begin, run the “man” command, or “vi”, or any command that requires a full screen. Then, while it’s running, press CTRL-Z. What happens?

_____

Next, use the “jobs” command. What does the output tell you?

_____

The command you have pressed CTRL-Z on is now ***suspended***. This means that it is “frozen”, and you can now type other commands. Repeat the previous operation with some other command, and likewise suspend it. Then use ‘jobs’.

_____

To return to the suspended jobs, use “fg %#”, replacing “#” with the job number.

_____

Now try the command ‘ls –lR /’. This is likely to start a listing of all the files on the filesystem – a very long listing, at that. Press CTRL-Z, and use the “fg” trick to resume the job.

Try the same operations, using “&” at the end of the command name. “&” sends the command to the background, and enables it to run concurrently. What commands fail the “&” and do not run in the background?

_____

_____

The UNIX Command line environment – Cont.

3. Introducing "screen"

Type the "screen" command. What happens?

_____

Not too impressive, is it? Type a few commands, or even enter a full screen application like "vi". Now, press CTRL-A, followed by 'c'. What happens? Try CTRL-A again, followed by SPACE. Can you explain?

_____

_____

Press CTRL-A followed by '?' to get the help, and find out what key sequence you would need to "lock" the screen.

_____

Next, press CTRL-A followed by "D". You should get a message saying "[Detached]". Then, run "screen" again with the "-r" switch. Explain what happens.

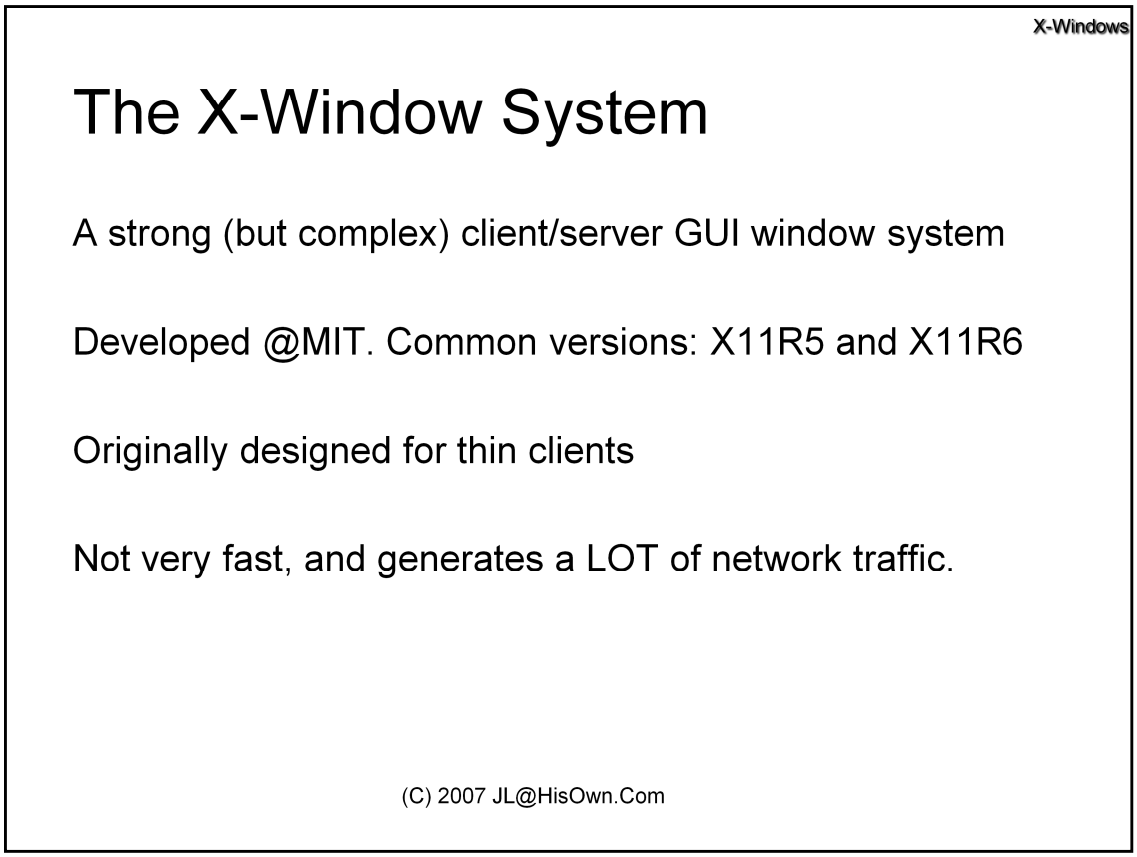
_____

Extra: start a different session from another terminal, while the first one is still in "screen". Then type "screen –r –D". What happens?

_____

_____

The X Window System



Long before Microsoft Windows, there was X. As Wikipedia points out:

"X derives its name as a successor to a pre-1983 window system called W (the letter X directly following W in the Latin alphabet). W ran under the V operating system. W used a network protocol supporting terminal and graphics windows, the server maintaining display lists. "

Other sources put the X in X-Windows for Xerox. Anyway, X is a truly powerful user interface, and is in fact the first user interface to support "Windows". Well before Apple "borrowed" it for its own, after which Microsoft "re-borrowed" from Apple.

X-Windows

The X-Window Display

X-Windows maintains a client server architecture (over TCP/IP), enabling GUI applications to run detached from the server. (Unlike Windows RDP)

The Display, or Server, provides the GUI for the application.

The actual application may be remote

Communication is carried over TCP ports 6000 to 6063(!)

X-Font Server (xfs) usually communicates over TCP 7100

A fundamental concept in X-Windows is the “Display”. Programs can redirect their GUI to a display (usually by means of a command line argument (-display) or by a corresponding environment variable. The display is also called the X-**Server**. Note roles here are somewhat counterintuitive to traditional “Client/Server”.

Note the large number of ports used (all TCP). X-Windows is rather hard to configure and secure from a network perspective.

X-Windows

Windows & Window Managers

The X-windows desktop is called the root (:0) display

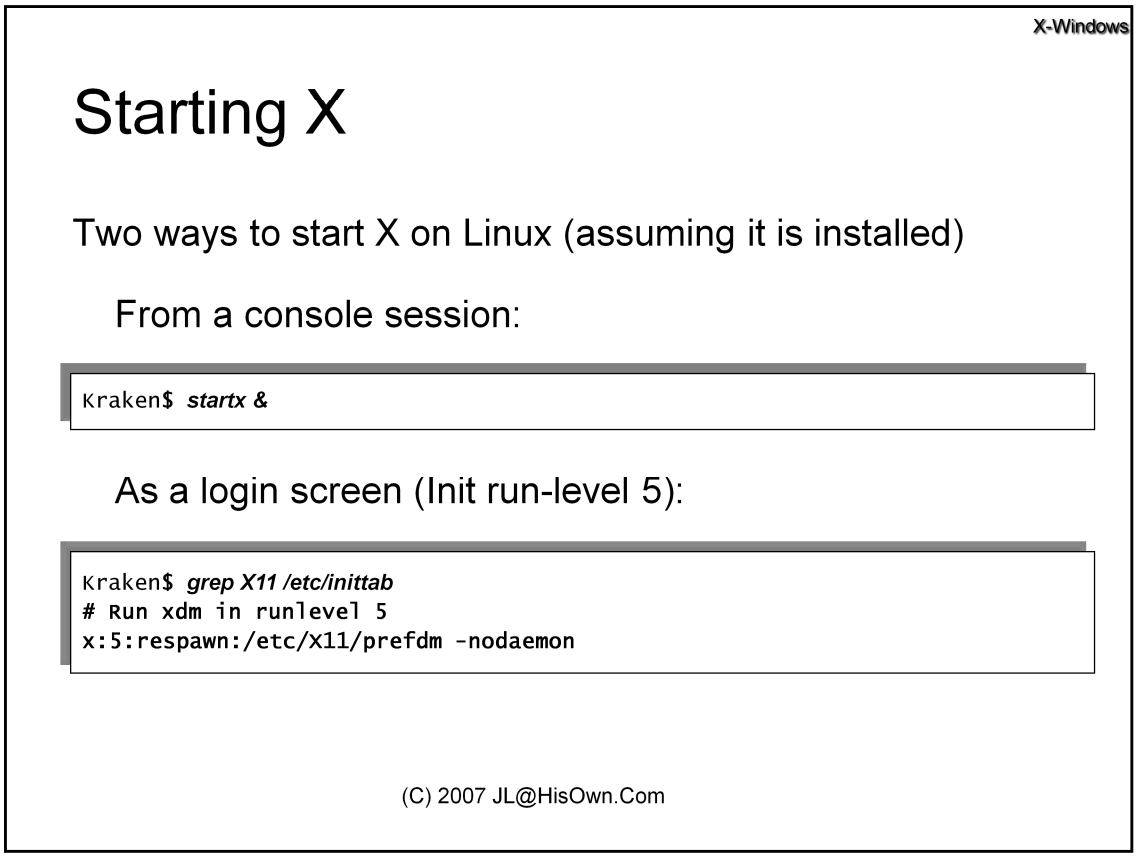
Windows from multiple clients may be redirected

Only client area is redirected.

A ***Window Manager*** frames and moves the windows.

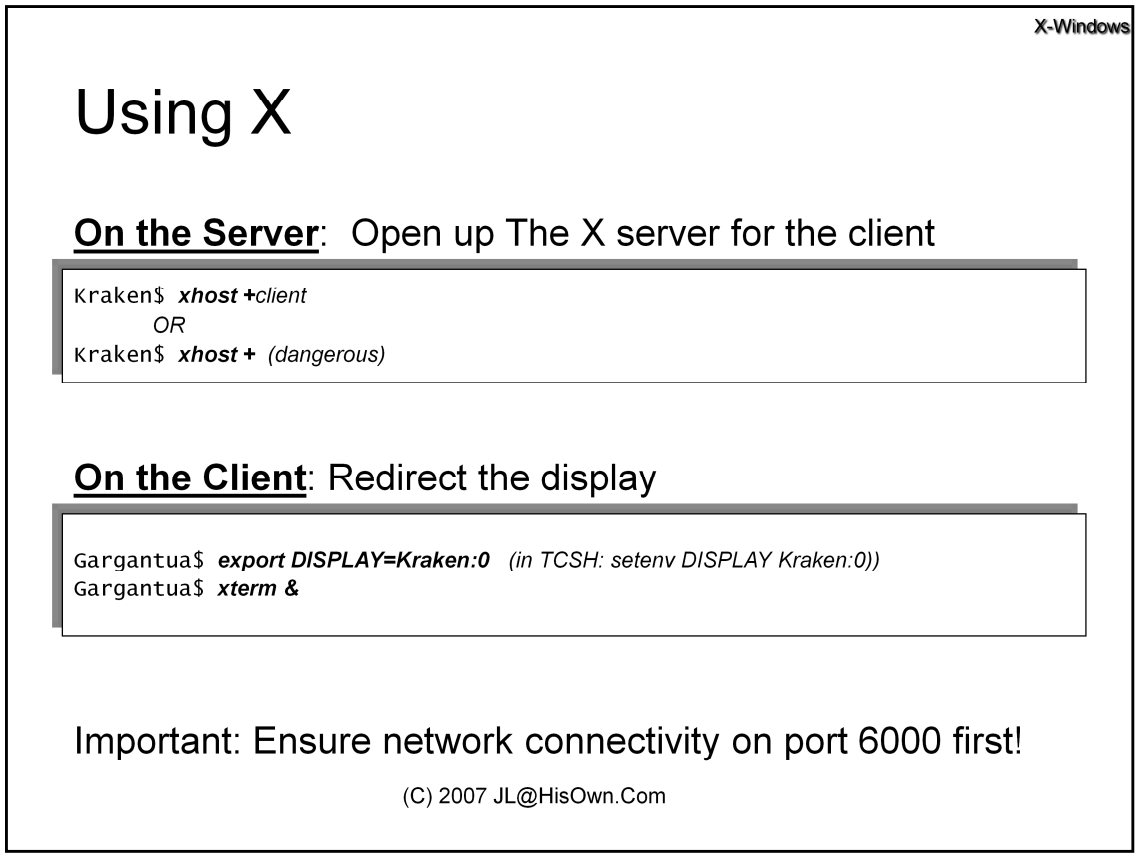
As stated, a very cool feature of X-Windows, still unmatched by Microsoft Terminal Server, is the ability to run just a single windowed application remotely, and not the entire desktop. In fact, an X-Windows session may host multiple applications from foreign clients. As windows are redirected, only their client area is passed. The rendering of the window frame and caption are left for a specific application on the server, called a ***Window Manager.***

Many such window managers exist. TWM, Motif, AfterStep, FVWM, and others. Linux provides Metacity (for GNOME) or KDE.



PrefDM is actually a redirector to either GDM (the GNOME Display Manager) or KDM (The KDE Display Manager)

Running in DM mode also allows other clients (most notably X-Terminals) to login graphically. The clients find the server by means of XDMCP – a (usually) broadcast based protocol.



X will not work if any firewalls filter out port 6000. To make sure that an X Session can be established, try to telnet to port 6000 manually first.

Important Note: Sometimes commands such as "netstat" will report connectivity and that port 6000 is listening – but X will not function. This is almost certainly because of the IPTables firewall service, or some other Kernel Level filter. In the case of IPTables, this could be disabled with /etc/init.d/iptables stop – although a better solution would be to add a rule to allow X-based traffic (ports 6000-6030, and port 7000).

X-Windows

1. Trying X-Windows

For this exercise, you will need to work with two computers. One (remotely) a client, and one (locally) a server. You can work on the remote client over SSH – so you don't *really* need to tie down two computers.

I. Login remotely to the client computer. Start an xterm remotely by means of the following two commands, illustrated in the slides:

1)_____ 2)_____

What happens? _____

II. How would you enable X to deal with the error message you've encountered?

_____

III. With the session active, run "netstat –n" on both machines. What do you see?

_____

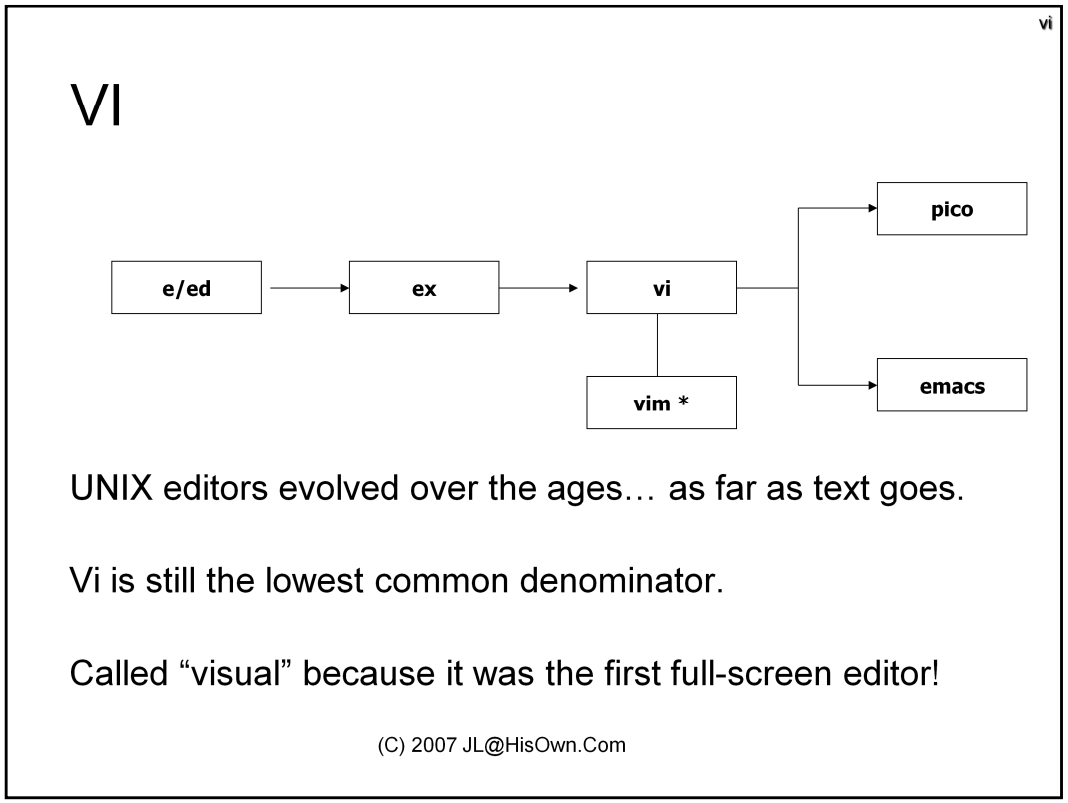
_____

IV. Repeat the exercise with a tcsh environment instead of BASH/ZSH. What is the major change required?

_____

_____

VI – The “visual” editor



Welcome to what just might be your worst nightmare – The wonderful “Visual Editor” – and the default editor in UNIX and Linux.

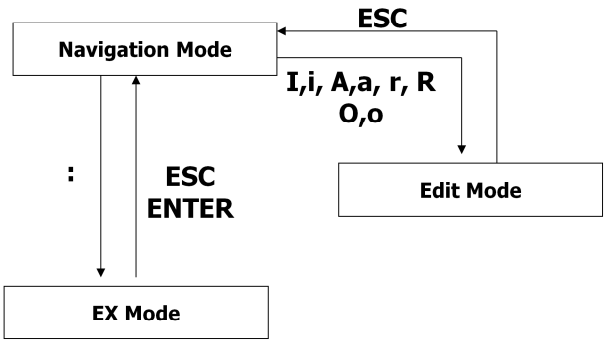
This “chapter” is going to be only a few pages in length – because it is primarily through the fingers that one learns how to use vi.

vi

The VI State Machine

- VI has to simultaneously allow:
  - Arrow-key/mouse like positioning in the file
  - A toolbar menu
  - A context menu
  - Editing

All in text mode!



(C) 2007 JL@HisOwn.Com

Tip: ALWAYS hit ESC-twice if you “get lost”. That will get you into Navigation mode, and a comforting “beep” will tell you that you’re “safe”.

Tip II: Careful with CAPS LOCK! This would CHANGE the meaning of your keys, therefore commands, and VI will go “crazy” on you. Especially when using h/j/k/l to navigate.. And they become H/J/K/L!

vi

Navigation Mode

- Commands in Navigation Mode (very, VERY partial list)
  - Arrows (up, down, left, right)
  - j,k,h,l (up, down, left, right)

Editing: (# = number of times to repeat operation)
- #x – delete (=cut) one character
- #d[d,w,L] – delete (=cut) line, word, or till end of screen
- #y[y,w,L] – yank (=copy) line, word, or till end of line
- p/P – paste deleted/yanked buffer after/before current line

---

Switch to edit mode:
- i/I – insert before current character/beginning of line
- a/A – Append after current character/end of line
- r/R – Replace this character or from this point on
- o/O – new line after/before current line

vi

Ex Mode

- Commands in EX Mode (fractional list)
  - :w (write) = save. Optional file name following
  - :x (eXit) = save & exit
  - :q (quit) = quit. Will prompt for save. q! to force
  - :s/regexp/regexp/ = Substitute (search/replace)
    - Requires line range. Use 1,$ for all file
  - :r – Read current file contents, or optional file name following
  - :## - Jump to line number ##

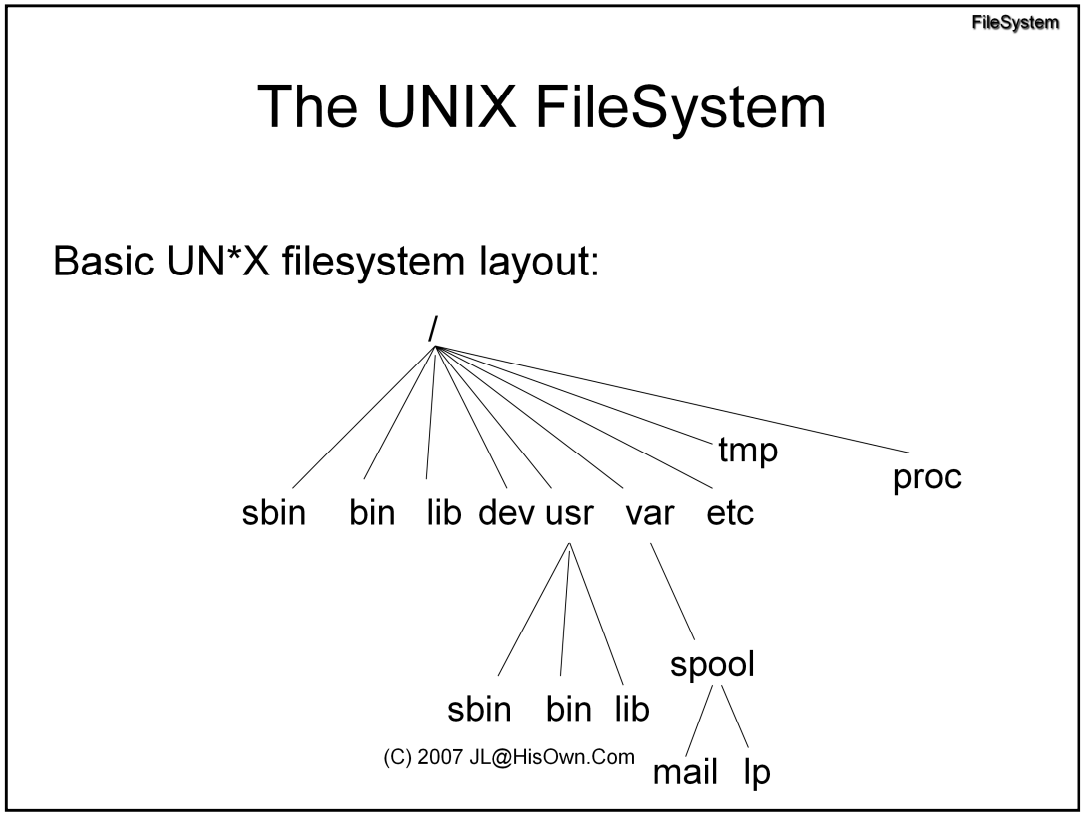
  - /regexp = Search for a pattern or regular expression

VI vs. VIM

Semi-standard “Vi IMproved”

- downloadable from http://www.vim.org
- supports syntax highlighting (syntax on)
- Language sensitive
- MANY many improvements over VI
- alias vi to vim if you can.

The File System, explained



All UN*X versions share the same basic file system structure:

- Files and folders are all under one root (/)
- Folders may either be local directories, or remote (NFS) mount points (much like windows network drives)
- The standard directories all have a well-known purpose.

Directory	Used for..
/bin	**"Binaries"** – i.e. executables, and standard commands (manual section: 1). This is where the core UNIX binaries was originally found: The basic subset of commands that UNIX cannot do command.
/sbin	**"System Binaries"** – executables that are used almost exclusively by root for system administration (those in manual section: 1M or 8). "Ifconfig" (the network interface configuration) is usually found here. /sbin is not in the normal user's path setting, so unless they specifically seek here, they're not likely to "see" any of these commands as available.
/lib	**"Libraries"** – UNIX, much like windows, has dynamic link libraries. Only here they're called "Shared Objects", and are marked with a ".so" extension. They are functionally equivalent to DLLs.

Directory	Used for..
/usr	**"User"** – This directory is used for 3rd party programs and additional software. (Think: "Program Files"). This directory contains, among others, a structure mirroring /bin/sbin/lib. The original idea was, that additional software would go here, so as to reduce chances of conflict (or accidents) with the core UNIX files. Over the years, however, many UN*X blurred the distinction between /usr/bin and /bin, to the point of actually linking them together. By convention, new software is usually installed either in "/opt", or in "/usr/local/bin", and the proper path modifications are performed in the users' login scripts.
/var	**"Various"** – This directory was used for various data files
/tmp	**"Temporary"** – This directory is used for temporary files. It has two important features: - **World writable:** Any user can read and write freely in this directory – not just root. Non root users, in fact, may find this is the ONLY directory they can write to, unless root has designated a home directory for them. - **Not persistent across reboot:** /tmp is not guaranteed to "survive" a system reboot, and is likely to be cleared. This led many UNIX vendors to implement /tmp as a "swap filesystem" – that is, in virtual memory. This makes sense because /tmp is frequently accessed – thereby improving overall system performance. Since virtual memory is cleared during reboot, this also achieves the "bonus" of starting with a clear /tmp every time.
/etc	**"Et Cetera"** – Latin for "all the rest", this directory started its life as a dump for "everything else" that couldn't find a place in other directories. It is used primarily, however, for configuration files. This is the closest to a Registry UNIX will ever get.

FileSystem

The UNIX Filesystem

UNIX only has a SINGLE mount point: / (root)

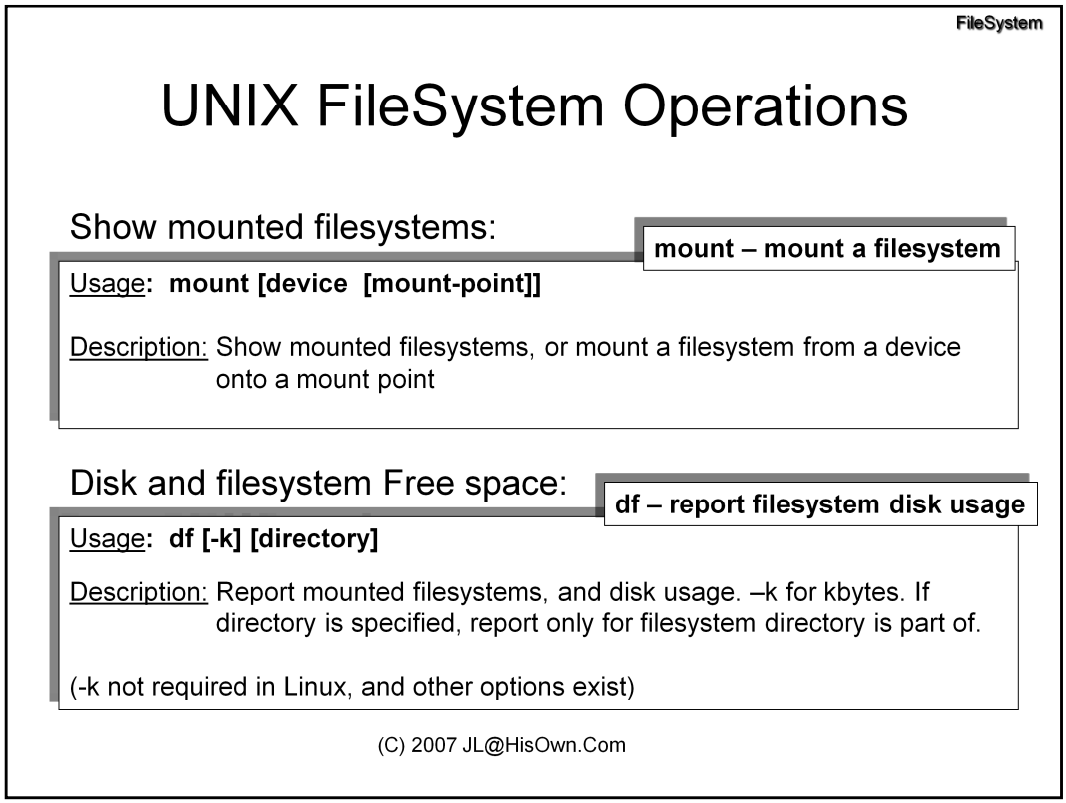
Multiple filesystems may be joined, or mounted, to the root

- 'mount'ing grafts the new filesystem onto the tree
- Any directory can be used as a mount point
  - If directory already has files, they are hidden by the mount.
- Filesystems can be:
  - Local: additional partitions on the physical disks
  - Remote: "network drives" exported via NFS or SMB

A key concept in UNIX filesystems is "mounting" – The method by means of which additional filesystems plug in to the existing filesystem hierarchy. "Mounting" is connecting a filesystem to an existing directory – that is otherwise normal, but is redefined as a "mount point". CD'ing to that directory will transparently move the user or process to the filesystem. Much like a "C:" "D:" would move in Windows.

Mounted filesystems are generally physical devices – partitions on disks. They can be easily identified because they read /dev/… for the device, usually /dev/hd* /dev/sd* or /dev/dsk… However, with the same ease and transparency, UNIX supports a distributed filesystem called "NFS" (The Network FileSystem), and Linux in particular also natively supports SMB – The Server Message Block that is the groundwork for Windows' filesharing (Think \\SERVER\SHARE).

When a directory is used as a "mount point", any files that it might actually contain are hidden. That is, hidden, not erased. Once the filesystem is unmounted, the files "magically" reappear.

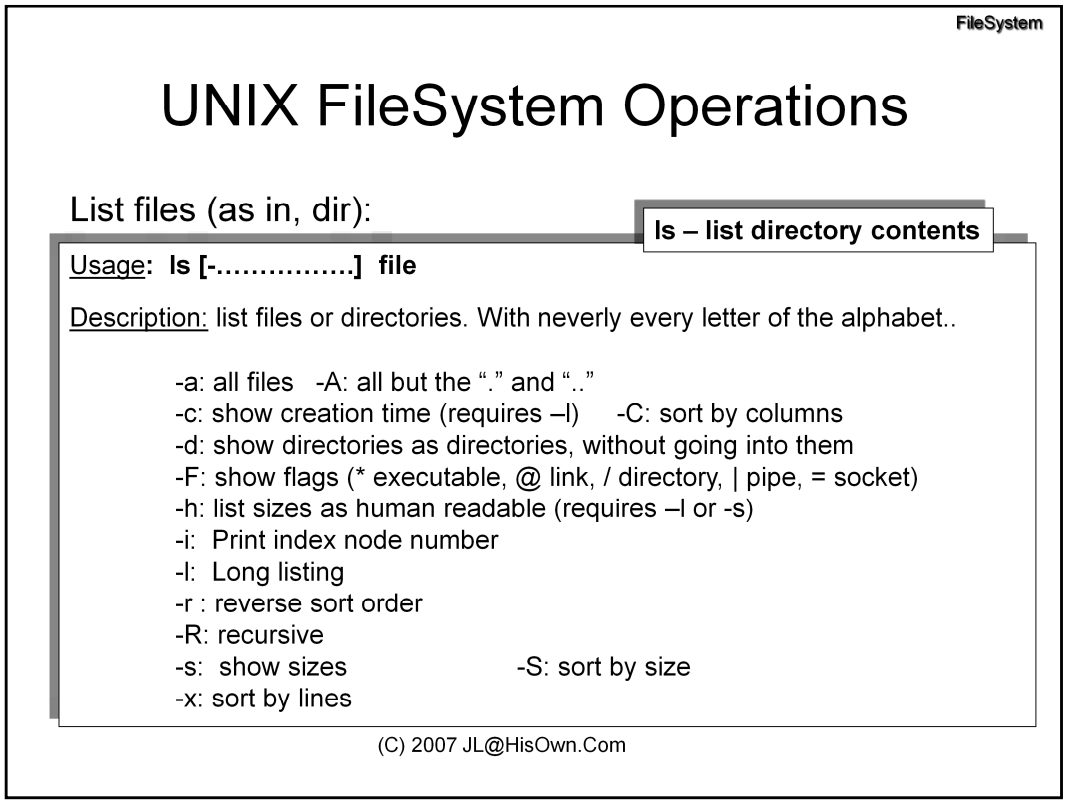


The "mount" command is used – not surprisingly – to achieve the mount operation. This command can only execute and actually mount a filesystem as the root user or administrator, and mounting is beyond our scope. But it still proves to be a useful command to see what filesystems are mounted, on which mount points.

```
Kraken$ mount
/dev/sda4 on / type ext3 (rw)
/dev/proc on /proc type proc (rw)
/dev/sys on /sys type sysfs (rw)
/dev/devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda1 on /boot type ext3 (rw)
/dev/shm on /dev/shm type tmpfs (rw,noexec,nosuid)
/dev/sda3 on /tmp type ext3 (rw,noexec,nosuid,loop=/dev/loop0)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
Jormungand:/tmp on /jormy type nfs (rw)
```

Similarly, "df" shows a list of mounted filesystems, along with their usage. Notice that this shows less filesystems than "mount", as it only related to physical devices:

```
Kraken$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda4            147237696  49276096  90361652  36% /
/dev/sda1               202220     78350    113430  41% /boot
/dev/shm                511304         0    511304   0% /dev/shm
/dev/sda3              1976268     20800   1853456   2% /tmp
Jorumngand:/tmp     143490833  481023331 90519102   35% /Jormy
```



'ls' is the most well known and one of the most versatile UNIX commands. It lists files and/or directories, with a surprsing amount of switches – nearly all letters of the alphabet are supported (and case sensitive!)

In Linux, a VERY common option (often aliased) is '--color=tty'. While this option is far from the standard, it gives the very noticeable effect of file listings in color.

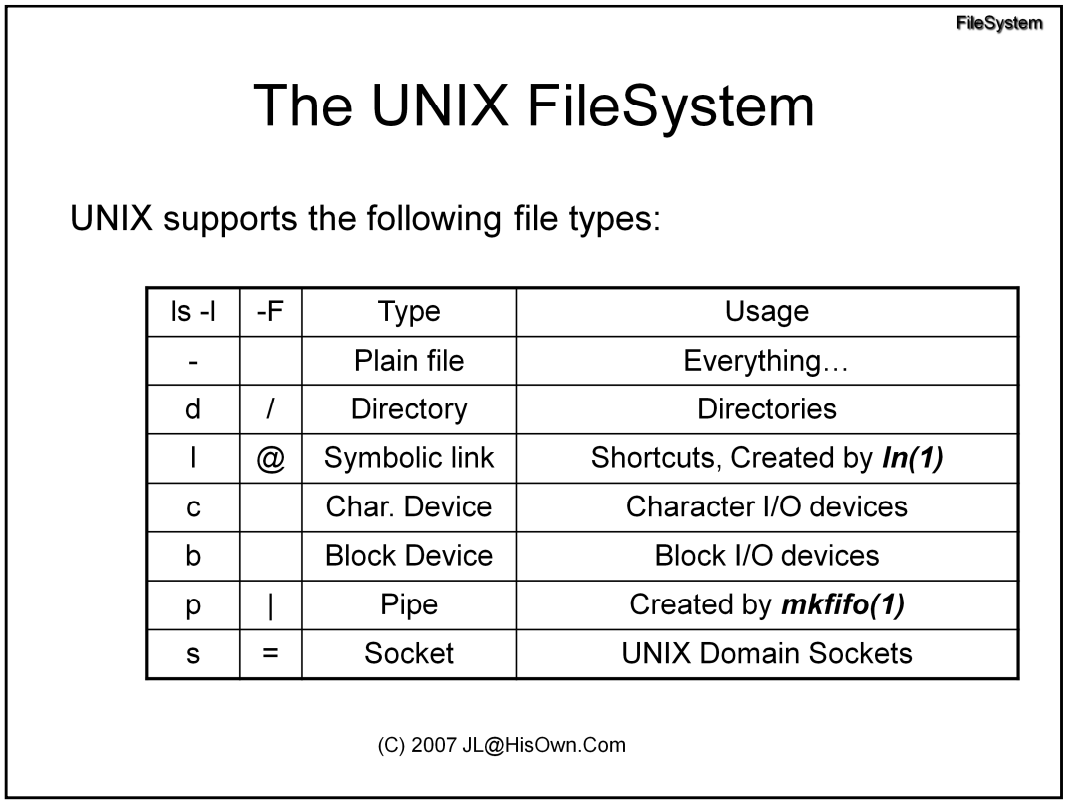
Unlike DOS, UNIX keeps three times for each file:

Access Time: Time of last access, read or write, unless the filesystem is mounted with "noatime" option

Creation Time: Time of file creation.

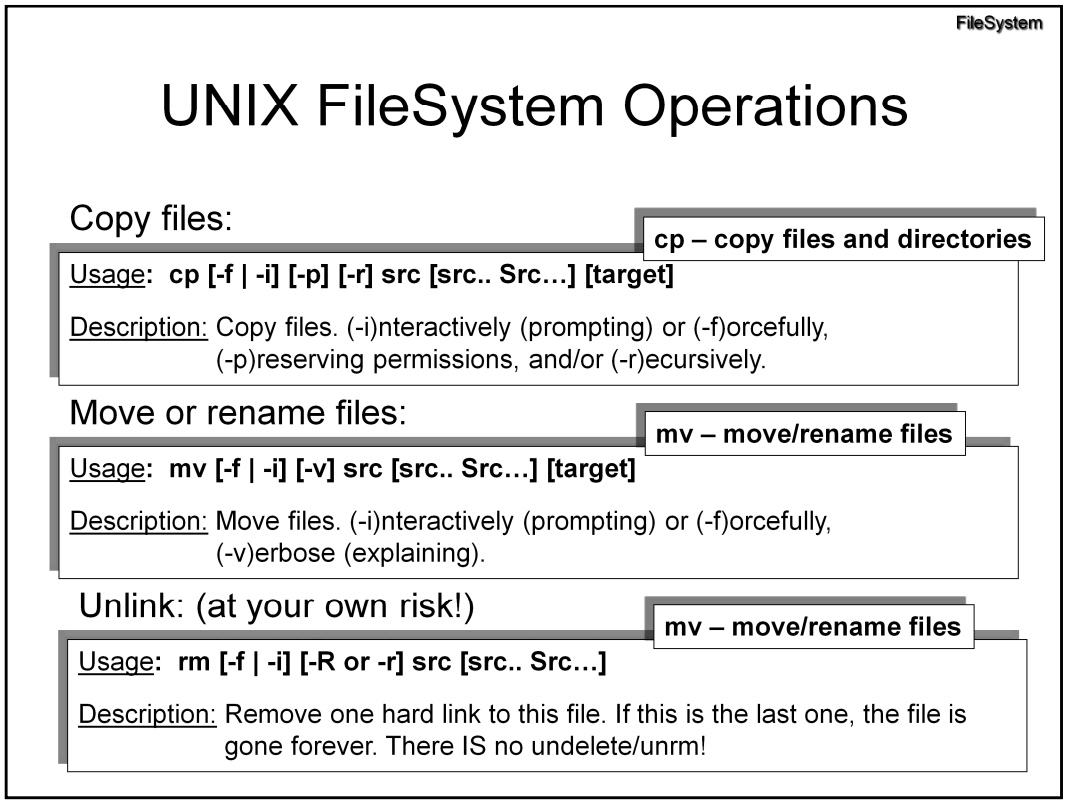
Modification Time: Time of last write operation.

These times, however, can and often are manipulated, using the 'touch' command. Touch may change any of the timestamps by –a –c or –m, respecively.



The above table shows the various types of files in UNIX and Linux, as shown by the 'ls –l' and 'ls –F' commands.

The important thing to remember is that EVERYTHING is represented as a file. (including devices – more on that later). Pipes and Sockets are the UNIX IPC mechanisms.



'cp', 'mv' and 'rm' are three of the most commonly used commands, as they enable the moving, renaming or copying of files around the UNIX filesystem. The syntax is very similar, as shown above. Their usage is straightforward.

Note: for novice types, consider aliasing "cp" , "rm" and "mv" with their safer "-i" switch… especially rm, especially if you run as root!

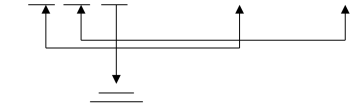
Note II: Be CAREFUL with Recursive use of 'rm', especially with the "*" wild card. "rm –fR *", a common "deltree" command, can wipe out your entire filesystems (ALL of them) if executed by mistake from the root!

FileSystem

UNIX File Permissions

ugo where no man has gone before

```
root@Paragon (/) #ls -l `which ls`
-r-x--x--x    1 root     root       24634 Jul 12  2000 /usr/bin/ls*
```



Permissions can be set for READ (r), WRITE (w), or EXECUTE (o)

In shorthand notation: r – 4, w – 2, x - 1

Permissions are defined for the USER (u) – file owner, GROUP (g) – one group, OTHERS (o) – ("world").

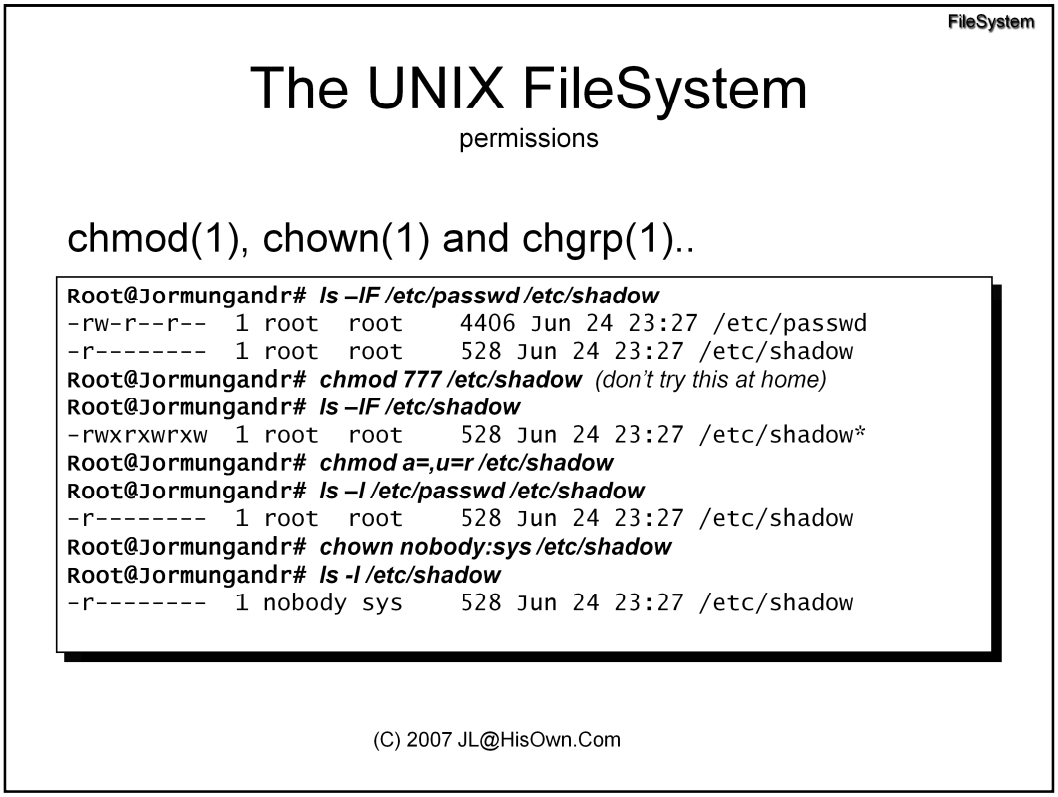
In a very primitive and naïve model.

Owner permissions are meaningless (may be chmod'ed at any time)

(C) 2007 JL@HisOwn.Com

The following table summarizes permissions, in their octal. As the table shows, it's much easier to get used to the octal notation, rather than work with –r, -w, and –x.

Bits	Octal	LS displayable
000	0	--- (= or -rwx)
001	1	--x (=x or -rw+x)
010	2	-w- (=w or -rx,+w)
011	3	-wx (=wx or -r,+wx)
100	4	r-- (=r, or +r,-wx)
101	5	r-x (=rx, or +rx,-w)
110	6	rw- (=rw, or +rw,-x)
111	7	rwx (=rwx, or +rwx)



chmod (Change Mode) changes permissions,

chown (Change Owner) changes the owner of a file,

chgrp (Change group) the owning group.

FileSystem

The UNIX FileSystem

Links

UNIX filesystem links are created with ***ln(1)***

```
Root@Jormungandr# ls –l /etc/passwd
-rw-r--r--  1 root  root    4406 Jun 24 23:27 /etc/passwd
Root@Jormungandr# ln /etc/passwd /hlink2Passwd
Root@Jormungandr# ls –l /etc/passwd /hlink2Passwd
-rw-r--r--  2 root  root    4406 Jun 24 23:27 /hlink2passwd
-rw-r--r--  2 root  root    4406 Jun 24 23:27 /etc/passwd
Root@Jormungandr# ls -i passwd /hlink2passwd
40424 /link2passwd    40424 passwd
Root@Jormungandr# find / -xdev -inum 40424
/etc/passwd
/link2passwd
Root@Jormungandr# ln  /etc/passwd /tmp/hlink2passwd
Root@Jormungandr# ln –s /etc/passwd /tmp/slink2passwd
ln: invalid cross-device link
Root@Jormungandr# ls –l /etc/passwd /slink2Passwd
lrw-r--r--  1 root  root    4406 Jun 24 23:27 /slink2passwd ->
                                                 /etc/passwd
```

UNIX supports two types of link files – and the two are totally unrelated.

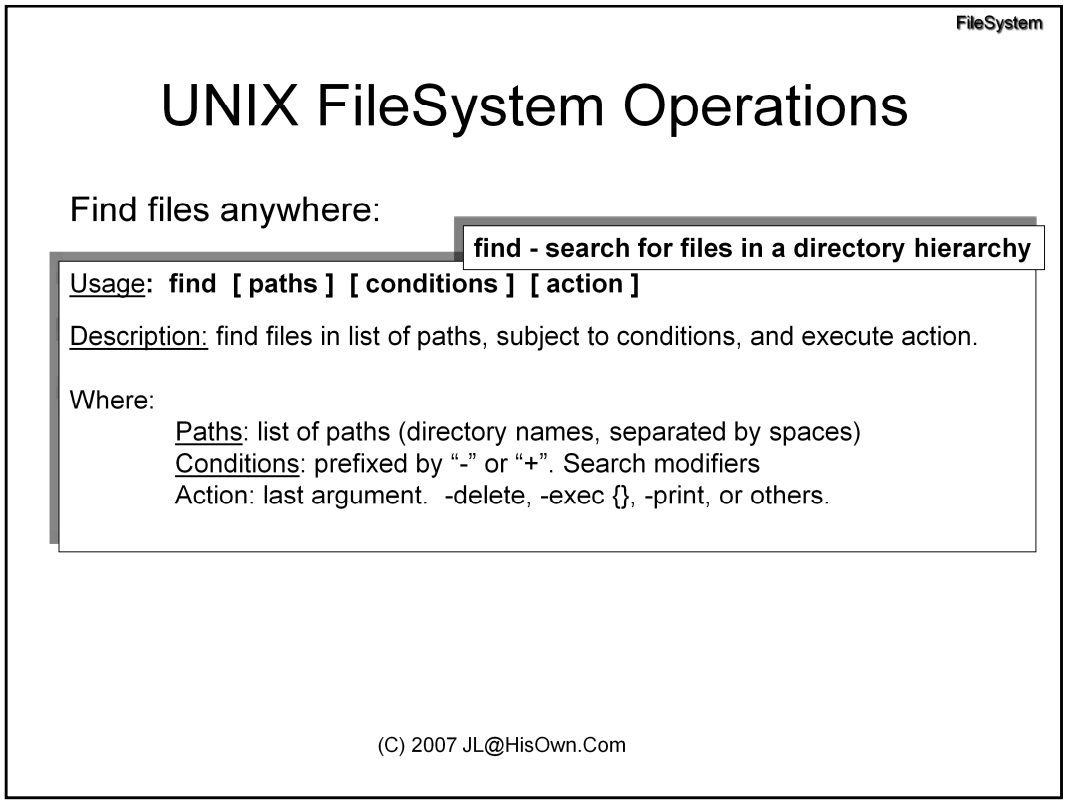
Both are created using ***ln(1)***. The default are hard links. For soft/symbolic, use "-s".

Hard Links: Another pointer to the file.

- Linked at the INODE level: this is NOT a separate file, but, rather – a separate name for the same file.
- Appear as same file. Link count in ***ls –l*** reveals hard link. ***find –inum*** finds all links.
- Cannot traverse filesystems, as inodes are only unique per filesystem.
- If target is deleted, contents unscathed

Symbolic Links: **Another name to the name of the file.**

- Two DIFFERENT files – the link is a NEW file, whose contents are merely the name of the target.
- Appear with a "@" in ***ls –F***, and "l" in file type, in ***ls -l***
- Can traverse filesystems, as name is resolved again each time.
- If target is deleted, link is broken.

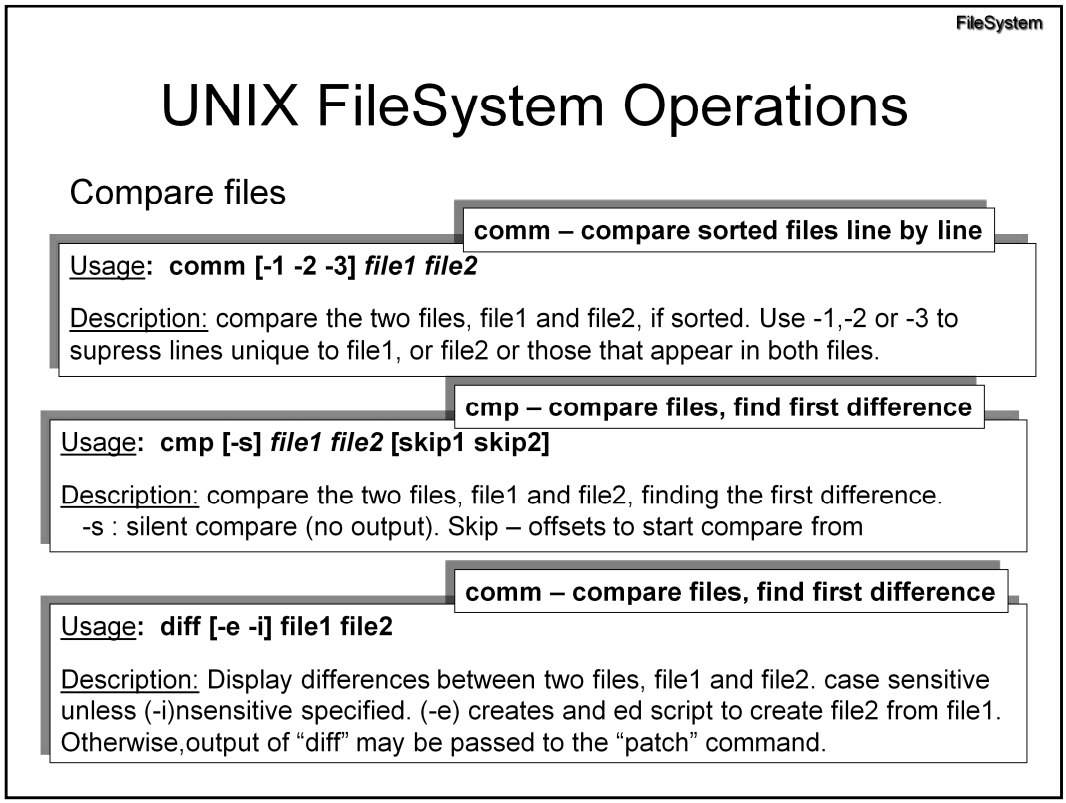


'Find' is a highly useful command to locate files on the filesystem according to a host of criteria.

condition	Meaning
-inum ###	Filename has inode number ###. Useful for finding hard links
-name ___	Find files with name ____. Specify wildcards * and ? in quotes
-size ….	Find files with size … . Use + to specify minimum, - for maximum. Also use G, M, K for Gigabytes, Megabytes, Kilbobytes
-perm ….	Find files with exact permissions … or also with permissions +
-type t	Find files of type t (f = file, d = directory, l = symbolic link…)
-newer/-older file	Find files that are newer or older than file.
-atime/-mtime/-ctime n	Find files with access/creation/modification time of n days ago Also: use +n or –n for minimum/maximum.
-user/-uid/-group/-gid	Find files that are owned by user or uid or group or gid

Once files are found, one of several actions may be employed:

Action	Meaning
-delete	Execute “rm” on the file, deleting it. (dangerous, but useful)
-exec …..	Execute any command, once per file found. Use {} to replace the filename in the command. (In most shells, use `\{\} \;`)!
-ok	Same as –exec, but prompt each time on command
-fprint *outfile*	Print name of file(s) found to file outfile
-print	Print names of file(s) found to stdout (this is the default action)



UNIX provides useful utilities to compare files and find differences.

Comm, cmp, and diff are similar, but sufficiently different to serve different purposes.

Diff is especially useful, since its output may be used by another utility, patch. Minor changes in huge files, like the Linux Kernel sources, come out in patch form, so they can easily be applied on a continuous basis.

FileSystem

SetUID Binaries in UNIX

How things REALLY get done

In UNIX, if you're not UID 0, you're a 0.

Problem: How do users perform privileged operations, like changing passwords?

Answer: They BECOME root, for the purposes of executing /usr/bin/passwd, or other such commands.

These are known as "setuid" commands (chmod u+s)

UNIX has a clear distinction between root, UID 0, the omnipotent super-user,

And all the other users, with a UID > 0, who can do virtually nothing.

However, to change one's password involves a write operation to the /etc/passwd

and/or /etc/shadow files… which can only be written to by the root user.

The same goes for scheduling jobs (at, cron), using low level or raw sockets (ping, traceroute), and other commands.

UNIX supplies a 'workaround' mechanism, in the form of SetUID.

FileSystem

SetUID Binaries in UNIX

It's all in that 's'..

```
root@Paragon (/) #ls -l `which passwd`
-r-s--x--x    1 root     root       13536 Jul 12  2000 /usr/bin/passwd*
```

Anybody with EXECUTE permission to a setuid binary, becomes, upon execution, equivalent to the owner
of that binary (in this case – ROOT)

(C) 2007 JL@HisOwn.Com

To make a program setUID, use:

chmod u+s program_name
or use a leading '4' in the octal mode, e.g.
chmod 4755 program_name

SetGID, that is, becoming a member of the group of owners upon execution,
is also possible, though less used.

chmod g+s program_name
or use a leading '2' in the octal mode, e.g.
chmod 2755 program_name

FileSystem

SetUID Binaries in UNIX

A source of insecurity..

Assumption: SetUID binaries are..

- **Sterile**: Write and manipulate well known files, under strictly predictable conditions.

- **Hermetic**: Users can't "break" the commands, to keep setUID privileges once command is done.

Unfortunately, sterile commands are far from hermetic, and vice versa….

Over 50 SetUID commands are usually found.

Naturally, SetUID programs are potentially disastrous.. If someone could execute a shell when under a setUID, he could get an instant "root-shell", in which any command issued would be effectively a root-issued command.

Find / -user root –perm +04000 –print

finds SetUID commands

FileSystem

Devices

- I/O follows UNIX “Device convention”
- Devices may be:
  - Character devices
  - Block (buffered) devices
  - Network Devices
- Devices normally id’ed by major and minor #

Linux supports the UNIX concept of a “Device”. All input/output to devices works as it normally would with a file, only the file – while present in the filesystem - is not a normal file on the disk, but rather – an interface to the device driver.

There are three types of devices, and we will deal with them all:

- Character Devices: Are devices which perform I/O on a “character” or unbuffered basis. Most devices in the system are indeed implemented as character devices. These include the memory, terminals, I/O ports, keyboard, mouse, and others
- Block Devices: Are devices which perform their I/O on a “block” basis – Blocks are usually 512-byte chunks, and are buffered by the system’s buffer cache. Block devices may further have filesystems built on top of them. Common block devices are the hard disks and mass storage devices.
- Network Devices: Are a third class of devices – implementing the network interfaces.

Character and block devices are created in the filesystem using the “***mknod***” command. e.g:

```
# mknod /dev/mydev c 254 0
```

To create a device called /dev/mydev, with major number 254 and minor number 0.

The MAJOR is the # of device driver registered in the Kernel

The MINOR is the # of device registered in the driver.

To see a device driver listing, cat /proc/devices. Common devices are listed in the following table:

Device Name	Description	Major	Minor
mem	Physical Memory	1	1
kmem	Kernel Memory	1	2
null	“Black Hole”	1	3
zero	Infinite stream of “\0”s	1	5
random	PRNG (blocking)	1	8
urandom	PRNG (non-blocking)	1	9
tty0	Current Virtual Console	4	0
tty1..tty6	Virtual Consoles (Alt-F1..F6)	4	1..6
ttyS0..3	COM1..COM4	4	64..67
tty0	Current (active) TTY	5	0
console	Physical Console	5	1
fd,	Process File Descriptors. SymLinked to /proc/self/fd	--	--
stdin,stdout, stderr	Process descriptors. /dev/fd/0..2	--	--

Filesystem

1. File system operations on directories

For this exercise to work, make sure the /tmp directory has the permissions drwxrwxrwx. Now, login as one user, cd to /tmp. and create a file with private permissions (rw-------). Login as another, and try the following operations:

a) reading the file
b) writing to the file using >>
c) renaming the file
d) deleting the file

Should any of these be successful? Are they? Explain!

_____

_____

Repeat the above, but only after setting the /tmp directory using “chmod +t /tmp”. What happens? What is the effect of the sticky bit?

_____

_____

Filesystem (Cont.)

2. Symbolic links

Create a file, say, /tmp/your_name and fill it with content.
ls –l that file.
Create a softlink and a hard link to this file.
ls –l all three (your file, the soft and hard link). What has changed?

_____

_____

Now, rm the original file, and ls –l the remaining ones. What do you see?

_____

_____

Now, recreate the original file, and fill it with new content. ls –l again. What do you see?
Cat all three, and explain the results:

_____

_____

Processes

Processes

A ***process*** is an instance of an executing program in CPU.

UNIX is a preemptive multitasking system, capable of running many concurrent processes

Each process has its own distinct attributes

Processes

Daemons

A ***daemon,*** like a windows "Service" is a process running "as part of the operating system" – no direct UI.

Daemons are characterized by:

- Lack of controlling terminal (tty = '?')
- No user input/output
- no STDIN/STDOUT/STDERR
- working directory is root.

Most operating system processes and services are daemons. These include:

-The task schedulers (atd and crond)
-The print spooler (lpd)
-The InterNET Daemon (inetd)
-Web, mail and FTP servers (httpd, sendmail (smtpd), ftpd, respectively)
-Kernel pageswapper (swapd)
-Miscellaneous RPC servers (rpc.____d)

Viewing Processes

/usr/bin/ps (sysV)

```
usage: ps [ -aAdeflcjLPy ] [ -o format ] [ -t termlist ]
        [ -u userlist ] [ -U userlist ] [ -G grouplist ]
        [ -p proclist ] [ -g pgrplist ] [ -s sidlist ]
  'format' is one or more of:
        user ruser group rgroup uid ruid gid rgid pid ppid pgid sid taskid
        pri opri pcpu pmem vsz rss osz nice class time etime stime
        f s c lwp nlwp psr tty addr wchan fname comm args projid project pset
```

/usr/ucb/ps (BSD)

```
usage: ps [ -aceglnrSuUvwx ] [ -t term ] [ num ]
```

Non-standard: Solaris: prstat, others: top

The most useful command for viewing processes is, by far, ps. This command comes in two flavors, however – SysV (default in Solaris, and most other UN*X) and the BSD flavor (available in those systems as /usr/ucb/ps). Both have similar capabilities, although sometimes one is preferable to the other. A third version, by GNU also exists – with numerous options… Its features combine both the SysV and BSD versions:

********* simple selection ********* ********* selection by list *********

-A all processes	-C by command name
-N negate selection	-G by real group ID (supports names)
-a all w/ tty except session leaders	-U by real user ID (supports names)
-d all except session leaders	-g by session leader OR by group name
-e all processes	-p by process ID
T all processes on this terminal	-s processes in the sessions given
a all w/ tty, including other users	-t by tty
g all, even group leaders!	-u by effective user ID (supports names)
r only running processes	U processes for specified users
x processes w/o controlling ttys	t by tty

********** output format ********** ********** long options **********
-o,o user-defined -f full --Group --User --pid --cols
-j,j job control s signal --group --user --sid --rows
-O,O preloaded -o v virtual memory --cumulative --format --deselect
-l,l long u user-oriented --sort --tty --forest --version
X registers --heading --no-heading
********* misc options *********
-V,V show version L list format codes f ASCII art forest
-m,m show threads S children in sum -y change -l format
-n,N set namelist file c true command name n numeric WCHAN,UID
-w,w wide output e show environment -H process heirarchy

Solaris also has ***prstat*(1)**, whereas other systems have ***top*** (displaying the "top 20" processes). AIX has monitor, and HP-UX – gpm (a.k.a glancePlus)

Processes

Process Attributes

Attributes are visible using the various “ps” switches

Attribute	Description	/usr/ucb/ps	/usr/bin/ps
PID	Process Identifier	**x**	**x**
PPID	Parent PID	-l	-f,-l
CMD	Executing command	**x**	**x**
Priority	Execution Priority	-l	-l,-lc
NICE	Nice value	-l	-l
TTY	Controlling Terminal, if any	**x**	**x**
Owner	[e]UID, [e]GID of owners	-u	-f,-l
PGID	Process Group Leader PID		-j
SID	Session Group Leader PID		-j

Every process is uniquely identified by the PID – Process ID. This is a unique runtime identifier (usually 32-bit), allocated in a monotonically increasing sequence (it can wrap around, but 32-bits are a LOT of processes). The process ID is used in the various process manipulation and control commands.

Every processes also has a PPID – Parent process ID. This is the PID of the process that spawned it. (User processes are spawned by the login shell). All processes start as fork()ed clones of their parent, then drift off. The lineage traces back to init (PID 1, the master process). Also, a unique connection exists between signal and child, which will be discussed later.

CMD – The command that spawned the process, with or without arguments.

TTY – The controlling terminal. Most processes retain a link to their controlling terminal, which can be used for keyboard signals. Should a process lose its terminal, tty is marked as “?”.

Owner – Every process is owned by a certain user. Most are owned by root, some by daemon, and the rest – by users. Root can control all processes (naturally). Other users are limited to control over their own processes.

Process groups and sessions groups are not discussed here.

Priority/Nice – All processes are created equal, but some are more equal than others.. The process scheduling priority gives some processes precedence over others. This is usually fixed, and cannot be modified (at least not by normal users). Users can modify the base priority to a limited extent, however, using the ***nice(1)*** command. Nice enables the user to be "nice" to others, and relinquish CPU time for others, by specifying a "nice value" – 1 to 19. The name dates back to the dark ages, when UNIX was run on very weak PDPs. Root can be not-so-nice, and specify negative nice values, 1 to -20.

Processes

Process Statistics:

Statistic	Description	/usr/ucb/ps	/usr/bin/ps
STATE	Execution State	**X**	-l
STIME	Time of execution	-u	-f
TIME	Cumulative time spent	**x**	**x**
SZ	Size in memory	-u,-l	-l
%CPU	Statistical Usage of CPU	-u,-v	
%MEM	Percent of Memory in use	-u,-v	

Process States:

O	Process is currently executing
R	Process is Runnable, and queued
S	Process is sleeping, or waiting for an event
T	Process is either sTopped, or Traced
Z	Process is a Zombie: also appears as <defunct> in the ps list.

STIME: The time this process first entered execution in CPU.

TIME: The cumulative time spent in the CPU, actually executing (that is, processing opcodes, and not sleeping).

SZ: Process size in memory.

Changing Priorities

nice:

```
nice: usage: nice [-n increment] utility [argument ...]
```

renice:

```
usage: renice [-n increment] [-i idtype] ID ...
       renice [-n increment] [-g | -p | -u] ID ...
       renice priority [-p] pid ... [-g pgrp ...] [-p pid ...] [-u user ...]
       renice priority  -g pgrp ... [-g pgrp ...] [-p pid ...] [-u user ...]
       renice priority  -u user ... [-g pgrp ...] [-p pid ...] [-u user ...]
  where -20 <= priority <= 19
```

(Solaris) priocntl:

```
usage:  priocntl -l
        priocntl -d [-i idtype] [idlist]
        priocntl -s [-c class] [c.s.o.] [-i idtype] [idlist]
        priocntl -e [-c class] [c.s.o.] command [argument(s)]
```

The ***nice(1)*** command, as stated, gives users the ability to relinquish their CPU time in favor of others. This is useful if the command is known a priori to be of low priority.

renice(1) works on already active processes, changing their nice value.

Root can, of course, specify negative nice values.

priocntl(1), in solaris, enables fine tuning of process priority. Process may be assigned to one of three classes:

- Real Time
- Time Sharing
- Interactive

Processes may have their priorities changed in real time.

Process and files

To find open files held by a process (Solaris) – pfiles:

```
Root@Jormungandr# pfiles
usage:  pfiles [-F] pid ...
  (report open files of each process)
  -F: force grabbing of the target process
```

To find processes holding specific files open – fuser:

```
Root@Jormungandr# fuser
Usage:  fuser [-[k|s sig]un[c|f]] files [-[[k|s sig]un[c|f]] files]..
```

fuser codes: c-cwd, m-mmap, n-lock, o-open,

r-root dir, t-text(code), y-terminal

Notice the output is rather cryptic: S_IFCHR stands for character device. S_IFDOOR – RPC door. dev (136,0) indicates the hard disk on which inode 996 (in this case, /devices/pseudo@0:4, tty4) resides. UID and GID are the owners of the file. Use find –inum to resolve the inode name.

```
Ymir$ pfiles $$
1081:   sh
  Current rlimit: 256 file descriptors
   0: S_IFCHR mode:0620 dev:136,0 ino:996 uid:1012 gid:7 rdev:24,4
     O_RDWR
   1: S_IFCHR mode:0620 dev:136,0 ino:996 uid:1012 gid:7 rdev:24,4
     O_RDWR
   2: S_IFCHR mode:0620 dev:136,0 ino:996 uid:1012 gid:7 rdev:24,4
     O_RDWR
Ymir$ cd /dev
Ymir$ pls -lLR | grep "136, *0"
brw-r-----   1 root     sys      136,  0 Jun 23 12:25 c0t0d0s0
crw-r-----   1 root     sys      136,  0 Jun 23 12:25 c0t0d0s0
Ymir$ find / -inum 996 -print
/devices/pseudo/pts@0:4
Ymir$ fuser /devices/pseudo/pts@0:4
/devices/pseudo/pts@0:4:    1352oy    1081oy    1079o
```

Processes

Signals

Signals are software interrupts – a semi-reliable mechanism for sending semaphore notifications to processes.

Signals are sent regularly by the O/S – but -

Users may send signals with the ***kill(1)*** command.

Signals are a very primitive form of process control – sending notifications to processes from other processes, or the kernel. The notifications are semaphores – that is, they indicate some event has occurred, but contain no data whatsoever.

The ***kill(1)*** command is a VERY useful command, enabling users to generate signals on their own. All 31 signal-types (on some systems, e.g. AIX, 64) may be sent manually using this command. Only some of them are commonly used, however, with the rest remaining obscure.

Signals

Useful Signals

Signal	name	Notes
(1) HUP	Hangup	Sent on terminal hangup; restarts daemons ; may be nohup'ed
2 (INT)	Interrupt	Equivalent of Ctrl-C.
3 (QUIT)	Quit	Equivalent of CTRL-\
9 (KILL)	KILL	Unmaskable, kills. 9mm to the head
15 (TERM)	Terminate	Default of kill(1). Graceful ending
16 (USR1)	User-Def1	User Defined Action 1
17 (USR2)	User-Def2	User-Defined Action 2

The table above lists the important signals.

HUP is sent by the terminal driver when the controlling terminal (tty) is disconnected (hence the name, hangup). Programs such as vi can catch the signal, save the session, and prevent loss of work. Most programs simply die. To enable programs to continue anyway, use the "nohup" command.

INT is sent by the terminal driver when the controlling terminal issues a CTRL-C (or the equivalent ***stty intr*** key). Most programs die, but some prompt the user to continue or abort.

QUIT is sent by the terminal driver upon a CTRL-\. Usually generates a core dump.

TERM is the graceful way to terminate a process – enabling it to catch the signal, and perform the custom rituals prior to departing the cruel world of UNIX. It is thus the default signal sent by the ***kill(1)*** command. Responsible programs close open files, commit or rollback transactions, and then clean up nicely. Others simply ignore this gentle plea, and go on executing.

KILL is the only unmaskable signal - the bona fide way to kill a process outright. Anything alive – dies instantly (no buts, no "in a second", no saving throw – in early UN*X versions, this killed init – and caused and instant kernel panic!). Of course, this also means no saving open files, and such. Due to programs ignoring the TERM signal, most administrators employ the bad habit of appending "-9" to the kill command. Be advised this IS dangerous – it works fine with stuck man or vi sessions, but you wouldn't want to kill your Oracle or Sybase with it. Therefore, Use at your own risk.

Some processes also accept **USR1** and **USR2** – one such example is named (the DNS daemon). Those that do, often use it for debugging. Most simply ignore it, however.

Processes

Signals

I/O related signals

Signal	name	Notes
23 (STOP)	Stop	Suspend (CTRL-Z)
24 (TSTP)	Terminal Stop	Sent to bg jobs on *tostop* or terminal exclusivity (full screen) request
25 (CONT)	Continue	Continue stopped/suspended jobs

Behavior controlled through:

- ***stty stop*** – followed by the suspend character (CTRL-V first)
- ***stty tostop*** – stop bg jobs on any output, not just full screen
- ***stty –tostop*** – undo stty tostop.

STOP and **CONT** are two of the most useful, yet unappreciated signals – the former suspends a process while executing (changing its status to sTopped), and the latter – resumes it.

When working in a local terminal, the STOP functionality is accessible via CTRL-Z (or ***stty susp***). The active process (also called a "job") is stopped (suspended) immediately. The CONT signal may then be sent either by typing "***fg***" (returning the job to the **f**ore**g**round) , or by "***bg***" **–** returning the job to the **b**ack**g**round. Jobs then execute merrily in the background until one of three happens:

- They need exclusive access to their controlling terminal (e.g. man, vi, full screen mode), or any input.

- They need terminal output (of any kind) when the ***stty tostop*** option has been set – then, a TSTP signal is sent.

- The controlling terminal exits, and they get HUP'ed. If that doesn't kill them, they are killed the first time they will require input from the terminal.

By sending these signals to a process on another terminal, one can simulate CTRL-Z and suspend processes (useful when you know a process is about to read/write from/to a file that isn't quite ready yet).

Processes

The less interesting signals

Signal	name	Notes
4 (ILL)	Illegal Instruction	Invalid opcode/mem. fault
5 (TRAP)	Debugger Trap	Useful for tracing
6 (ABRT)	Abort	Terrible error
7 (EMT)	Emulator Trap	Not really useful
8 (FPE)	Floating Point Exception	division by zero, overflow
10 (BUS)	Bus Error	I/O error
11 (SEGV)	Segmentation Fault	NULL pointers, memory fault, buffer overflows
12 (SYS)	Bad System Call	Invalid System Call (rare)
13 (PIPE)	Broken Pipe	"*Prog1* | *prog2*" – prog2 dies
14 (ALRM)	Alarm	"Alarm Clock"
18 (CHLD)	Child status changed	a.k.a SIGCLD. Child is dead.

The full list of signals is always available by "***kill –l***" , or "***man signal***" (section 3, (Solaris - 3HEAD)).

These signals are less useful (no real need to send them manually, other than during debugging). Most signals in this list simply terminate the process, with a core dump. The exceptions:

PIPE, ALRM – quiet exit

CHLD – ignored unless waiting for a child process.

Processes

Job Scheduling: at

To schedule jobs at a later time, use "at".
Jobs are executed in a batch mode, by the atd.

Output, if any, is mailed to the user (but may be redirected).

AT has two flavors: BSD and SysV. Linux follows BSD:

BSD	Sys V	Functionality
at	at	Add job to at queue
atq	at –q	Display at queue
atrm	at –r	Remove job from queue

(C) 2007 JL@HisOwn.Com

To run jobs in batch mode, use the ***at(1)*** command. This useful utility places commands in a queue, which is processed by the at daemon – atd. This daemon wakes up every minute and checks for pending jobs. If any are found, they are executed, with no controlling terminal.

When running batch jobs, it is important to make sure that they do not require input – use "<" and pre-defined answer files, if they do. Output, if any, is usually e-mailed to the user (how convenient ... ☺), but may be redirected to files with the ">" and ">>" operators.

While at comes in two flavors with different syntax, the functionality is essentially identical, and the same queue is processed in both cases.

Example:

Prompt> **at 17:30**

at> **echo "Go home!" > /dev/console**

at> **(control-D)**

Job Scheduling: cron

For recurring jobs, use the standard UNIX cron facility.
Jobs are stored in crontab files (/var/spool/cron/…)
accessible via ***crontab –e*** (or -l) command

```
Root@Kobold# crontab -l
#ident  "@(#)root       1.20    01/11/06 SMI"
#
# The root crontab should be used to perform accounting data collection.
#
# The rtc command is run to adjust the real time clock if and when
# daylight savings time changes.
#
10 3 * * * /usr/sbin/logadm
15 3 * * 0 /usr/lib/fs/nfs/nfsfind
1 2 * * * [ -x /usr/sbin/rtc ] && /usr/sbin/rtc -c > /dev/null 2>&1
30 3 * * * [ -x /usr/lib/gss/gsscred_clean ] && /usr/lib/gss/gsscred_cleanv
```

crontab –l : List crontab

crontab –e : edit crontab files with default EDITOR.

Crontab format is

Minute/Hour Hour/Day Day/Month Month/Year Day/Week Command

Multiple values may be separated by commas.

Processes

This exercise has you answering several process related questions

Processes and files

- Find the process ID of xinetd

_____

II. Using lsof, see which files are held by xinetd

_____

_____

III. Using the /proc file system, which command would you execute for this information?

_____

IV. Run 'vi' on some file in the /tmp file system. Next, in another tty, attempt to umount /tmp. Is it successful?

Which commands would you run to enable a umount of /tmp?

Process Control:

- Open two shells (in two separate ttys). Obtain their PIDs by "echo $$" in each.
- From one shell, send a STOP signal to the other shell. Then switch to it. What happens?

_____

_____

III. From the other shell, send a CONT signal to the other shell. Now switch to it. Any change?

_____

Scheduling Jobs – I - at

- Using the “at” command to set a job to capture all active processes in a minute.

_____

II. Using the “at” command set a job to kill all active shells one minute from now

_____

Scheduling Jobs – II – cron, anacron

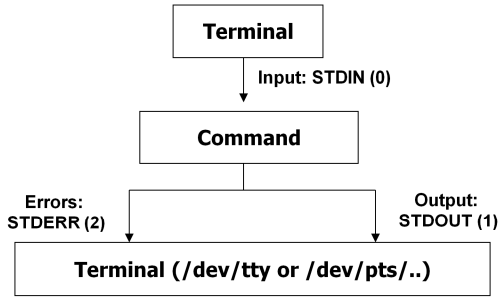
- Add a cron job to run 10 minutes after the hour.

_____

UNIX Filters

UNIX Commands as Filters

Most UNIX commands are designed as filters:



All streams are usually connected to terminal

UNIX was developed as a textual, terminal-oriented operating system. That far we already know. Functionally, this means that UNIX commands are suited to operate in a terminal – Taking input from files, or interactively from the user, and outputting messages and errors to the terminal.

From a design standpoint, program I/O is separated into three:

- **Standard Input**: Is defined as the stream from which input is read, when the program requires any input that is not explicitly requested from a file. By default, this "Standard Input" (or STDIN, for short), is connected to the user's terminal, as the user is the interactive controller of the command executed.
- **Standard Output**: is defined as the stream to which output is written. We say "output" to denote normal operational output, such as the results of execution, not including any errors, which are handled separately. Standard Output, (or STDOUT, for short), is likewise connected to the terminal, as the command needs to report back to the user, interactively situated at the terminal from which the command was invoked.
- **Standard Error**: is a *separate* stream that is used solely for the purposes of error reporting. Called STDERR, for short, it is normally indistinguishable from the output steram, as both are interleaved and connected to the terminal.

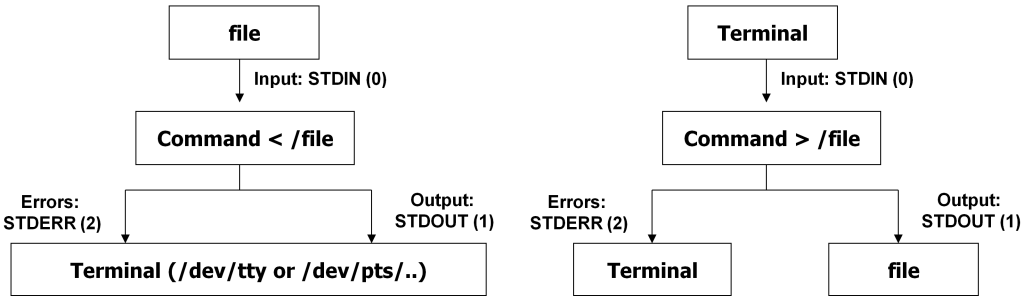
This behavior in UNIX is default in all CLI commands (but not some X-Windows commands). However, it generally goes unnoticed since (unless otherwise stated) all three streams – input, output and error – are connected to the terminal, providing the expected behavior of getting input from the user, and spitting output or error messages to the user. However, it is exactly this abilty to "otherwise state", or **redirect** the streams that provides UNIX with surprisingly flexibility in tying commands together.

While nearly all commands in UNIX can have their output and error streams redirected, filters prove to be a special subset. Filters operate either on files (if provided in the command line as a filename argument) or on their standard input. That is, unlike any other command, whose output may be redirected, filters allow a default mode for work on their standard input – thereby enabling them to READ input "prerecorded" in a file, or (as we shall shortly see), connected from another command. Filters also ***NEVER*** modify their input stream.

Redirecting Input/Output

All three streams can be redirected:

- STDIN can be connected to a file by means of the "<" operator
- STDOUT, likewise, by means of the ">" ("clob") or ">>" (append)
- STDERR – by means of the "2>" ("clob") or "2>>" ("append")



To demonstrate redirection, we will use a simple command – ls. This command, as you'll recall, provides a file listing. When called, however, with the name of a non existent file, it will issue an error message – after all, it cannot list that which does not exist.

If no redirection takes place, the error message is sent to terminal, along with the output of the file in question:

```
Kraken$ ls –l /etc/passwd nosuchfile
/bin/ls: nosuchfile: No such file or directory
-rw-r--r-- 1 root root 6223 May 13 02:31 /etc/passwd
```

However, notice the same, with the ">" operator, which we use to redirect the output stream elsewhere:

```
Kraken$ ls –l /etc/passwd nosuchfile > /tmp/output
/bin/ls: nosuchfile: No such file or directory
```

And with the "2>", which we will use* to redirect the error stream elsewhere:

```
Kraken$ ls –l /etc/passwd nosuchfile 2>/tmp/output
-rw-r--r-- 1 root root 6223 May 13 02:31 /etc/passwd
```

This can, and *will* prove to be ***very*** useful as we deepen our exploration into the UNIX realm.

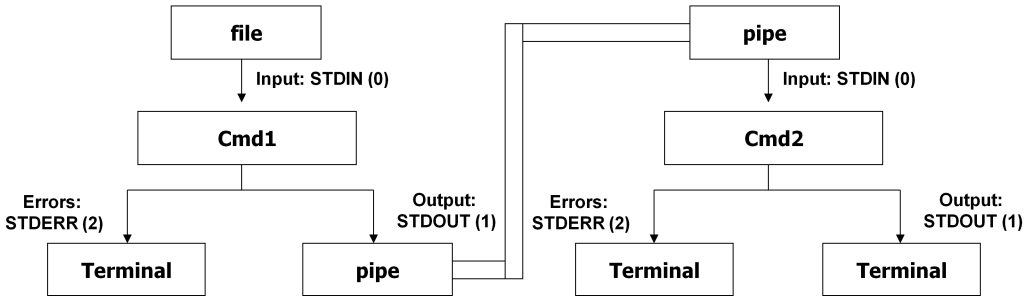
Certain caveats associated with redirection will be shown in the exercises.

* - We use 2> on all shells but those of the csh/tcsh variant. In those shells, error redirection is impossible without output redirection, and the syntax is somewhat convoluted.

Redirecting Input/Output

The STDIN and STDOUT streams can be piped:

- The "|" operator connects cmd1's stdout to cmd2's stdin
- The process can be repeated as many times as necessary
- STDERR can be independently redirected



Redirecting to files opens up a slew of possibilities for command expansion, by saving output from a command for later processing by another. However, more often than not the output of one command is useful as input for another, and commands are run in a direct sequence. In those cases, rather than doing something like:

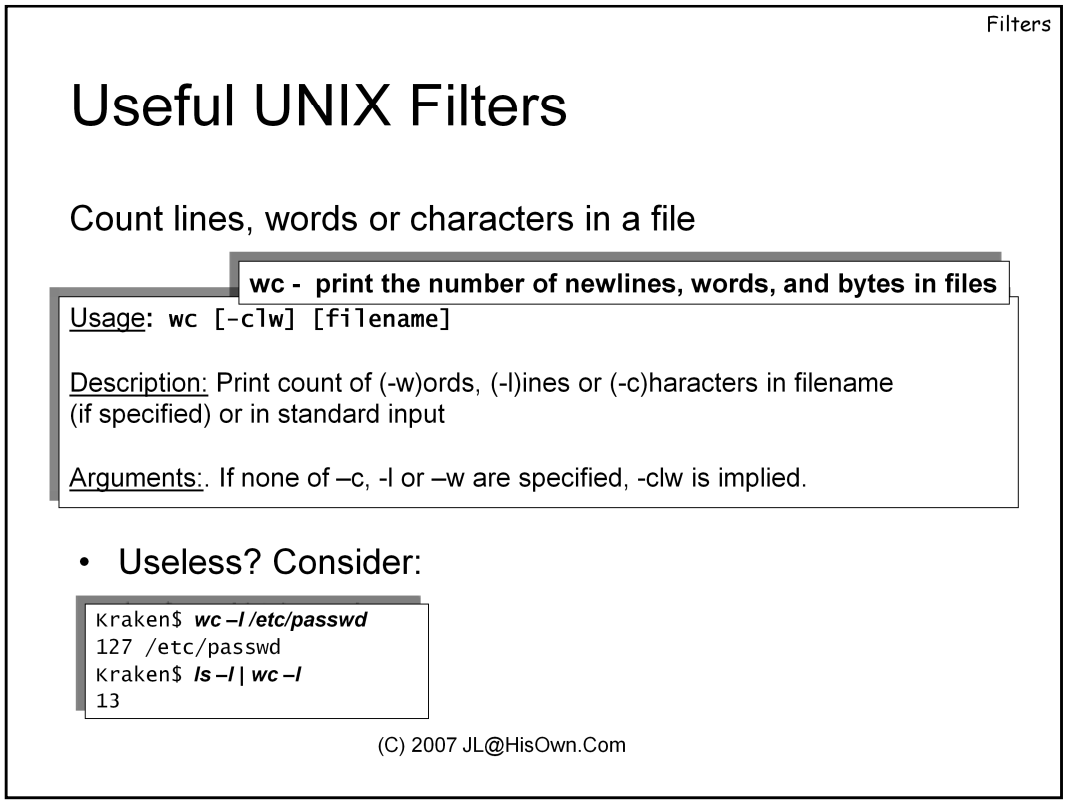
```
# cmd1 > CMD1_OUTPUT
# cmd2 < CMD1_OUTPUT > CMD2_OUTPUT
# cmd3 < CMD2_OUTPUT
```

And so forth, it makes more sense to *pipe* the commands together:

```
# cmd1 | cmd2 | cmd3
```

This powerful mechanism opens an in-memory channel over which the output stream of one command can be redirected into the input stream of another. Not just saving time for the user typing the sequence, but also increasing performance and saving on temporary disk space.

The fact that "pipelines" like these can be put to any length necessary makes UNIX commands highly extensible – but leaves some design work for its users, who now become architects, of sorts. UNIX provides you with the raw tools, that perform the most generic operations. It is up to you to craft more refined tools, suited for particular purposes, by joining the basic building blocks together. The next pages will show you these building blocks in action, individually as well as together with others.

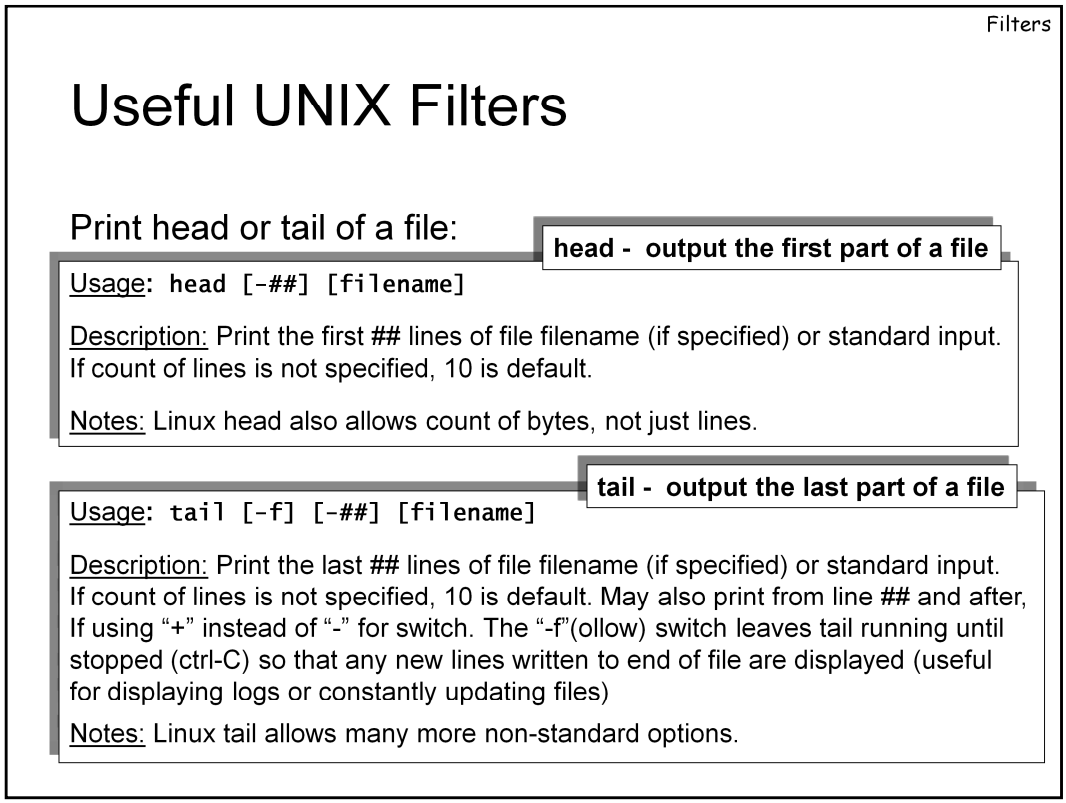


The 'wc' command (word count) is a generic utility that performs a very simple operation: counting words, lines or characters in a filename or its standard input. As a filter, it can work with either – defaulting to standard input if the filename is not specified.

While it might seem less than exciting behavior, it turns out to be useful behavior after all. The wc utility can serve to answer questions beginning with "how many", and have interpretation in file or line form.

For example – how many users are defined on the system? As the example shows, **wc-l /etc/passwd** can answer that question, if we take into consideration each user is represented as a line in that file.

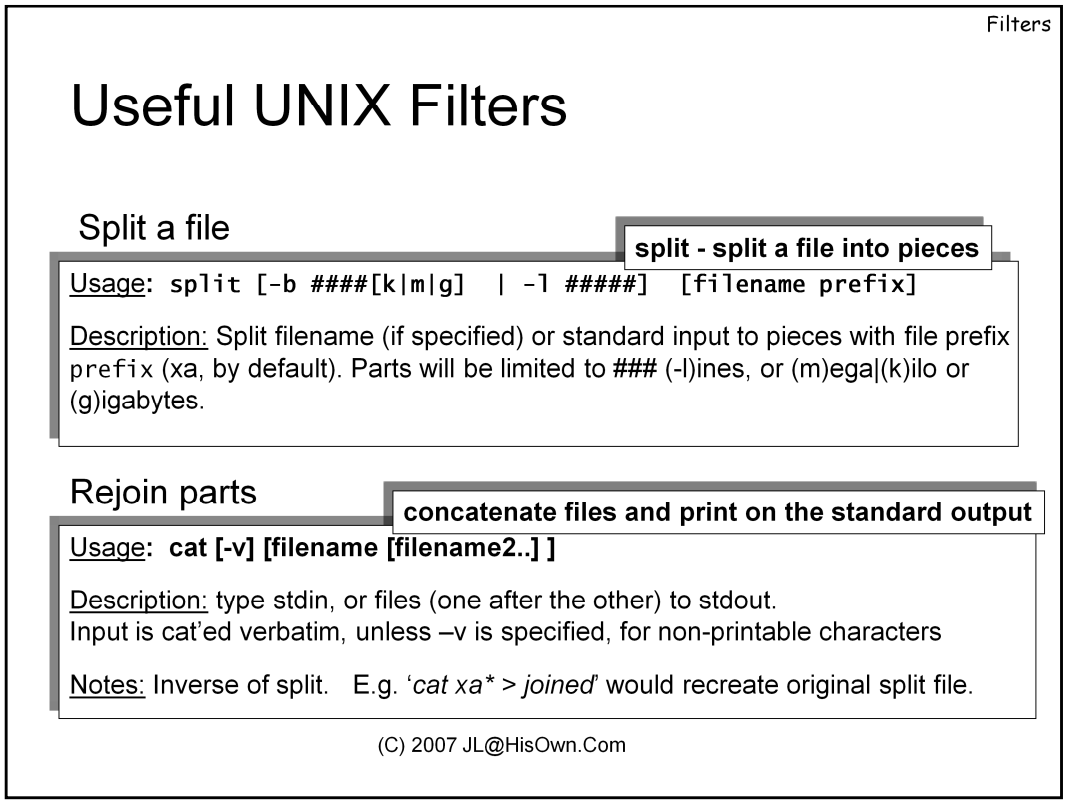
Another example – how many files in a particular directory? By counting the output of ls –l, (and remembering to substract one) for the "total" line, we can answer that: **ls –l | wc –l**.



Useful complements to "wc" are the "head" and "tail" filters. They allow the selection of lines from standard input or a file by line number. Their argument is a number (represented about by ##), which is the count of lines returned. If this argument is not specified, the default is 10.

Tail offers another feature of using "+" instead of "-". So that "tail -5" is the last 5 lines of the file, and "tail +5" is lines from the 5th and up (till the end of file).

Another useful feature is **tail –f** (thanks Craig!). This enables us to look at the last 10 (or other) lines of the file, and continue to hold the file open. Meaning, as new lines appear in the file (say, as output of messages to a log, or such), tail will display these lines as they become available. This is particularly useful, if used in conjunction with "grep", which enables one to isolate only those lines that are meaningful.

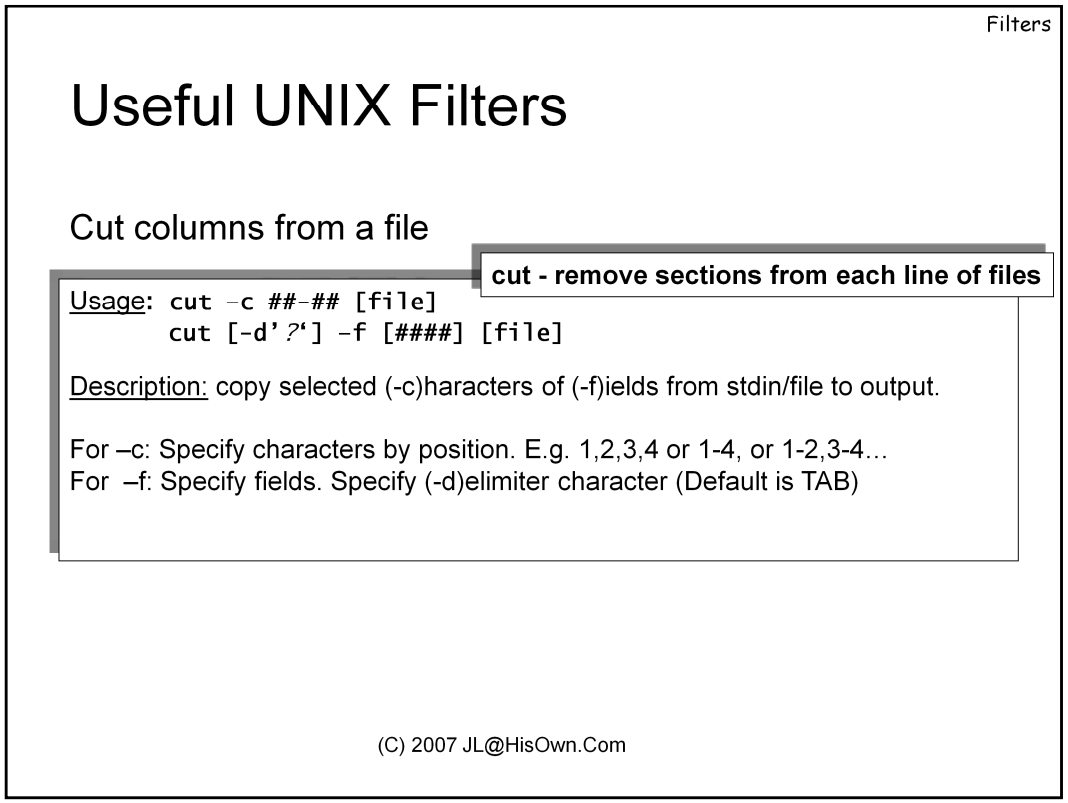


Picking up where "tail" and "head" leave off, "split" enables to cut a file (or standard input) to pieces. This is very common in breaking up huge files into manageable pieces (as was the case for a long time when floppy disks were used).

Split will automatically split its input into file named xaa through xaz.. (going to xba…. and further if necessary). An alternative prefix can be used – but since the prefix is the second argument, if you want to use stdin (as in, leave the filename argument blank), "-" is used.

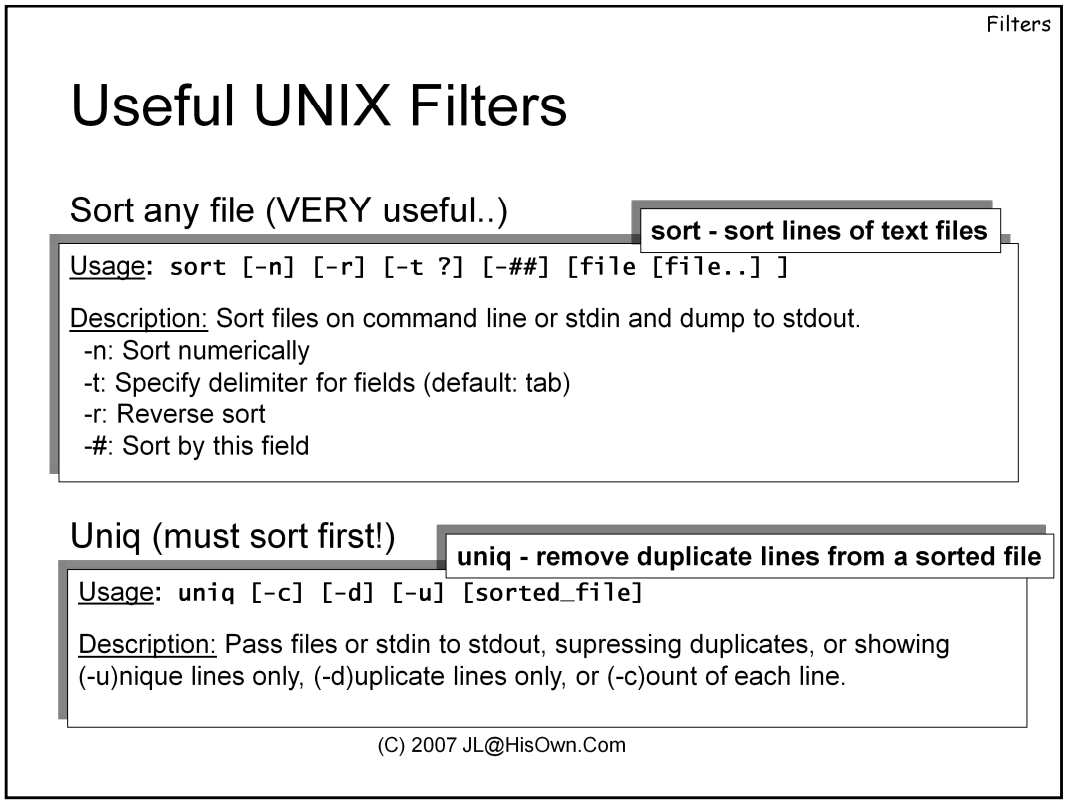
The switches to split are –b and –l (-b is supported everywhere, whereas –l started its life as another Linux extension). –b is very useful in that it can cut into specific file sizes in bytes (default), Kilobytes (by appending a "k" to the number), Megabytes (appending an "m") or even Gigabytes ("g"). The only drawback is that all "splits" have to be the same size.

To join files together, use "cat". Cat is also used to just type files to stdout.



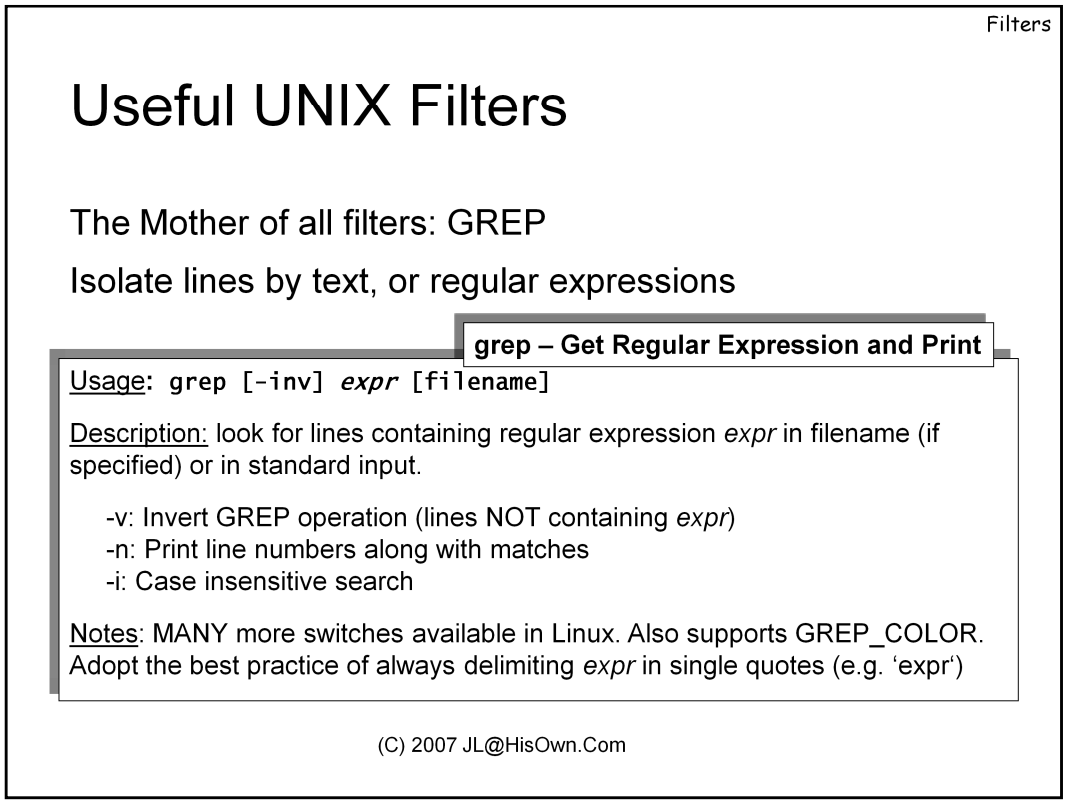
'cut' is a surprisingly useful utility that can isolate specific columns or fields from its input text, and pass only them to the output.

Cut operates either by characters (-c, in which case the character range should be specified), or by fields (-f) in which case the default delimiter for fields is TAB. This delimiter could be changed easily by specifying –d.



Sort and Uniq are two commands that are often used together, and the latter requires files to be sorted.

Sort serves as a universal input sorter on any text file type. The default sort is lexicographic, but can be modified by numerous switches.



Grep is, by far, the most powerful of all UNIX filters. This is due to two reasons:

- Grep answers a need for a common task: isolating lines in a file according to a certain text they contain. Whereas 'head' and 'tail' could find lines by number, grep enables you to look for keywords and other interesting identifiers in files or command output
- Grep bolsters the above functionality with ***regular expressions*** – these are pattern strings, that enable you to specify either fixed text, or use a variety of wildcards and placeholders.

If you are unfamiliar with regular expressions, or "RegExp"s, as they're affectionately known, you've been missing out. Regular expressions are *incredibly* powerful ways to deal with all forms of text. Instead of complex string operations such as cutting, splicing and concatenating, regular expressions can achieve the same functionality with a single pattern, that can be supported by many UNIX utilities (grep being just one of them – 'vi' is another), as well as modern programming languages. The appendix lists some regular expression syntax highlights.

The following examples show three modes of usage of this wonderful* command.

Example I: using grep for basic string matching

```
Kraken$ export GREP_COLOR
Kraken$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin
```

In the first example, we look for lines containing 'root' anywhere in the line, by simply specifying 'root'. The lines returned show where 'root' was found. We use the non-standard Linux extension of "GREP_COLOR" to highlight the results.

* - Yes, I'm biased, but no, I'm not getting carried away. Grep is a practical swiss army knife in the UNIX world, used for all sorts of purposes: from parsing log files to extending other commands. It also has ports to the Win32 world.

Example II: using '^' and '$' to limit matches

```
Kraken$ grep ^root' /etc/passwd
root:x:0:0:root:/root:/bin/bash
Kraken$ grep 'sh$' /etc/passwd
morpheus:x:500:500:Morph:/home/morpheus:/bin/zsh
```

Notice the difference in results for the second example illustrated. By prepending the "^" character to our expression, we now see only those lines that **begin with** our expression. '^' (shift-6 on most keyboard) thus serves as an imaginary 'beginning-of-line' character. Likewise, the '$' serves as an imaginary 'end-of-line' character, and appending it to the expression gives it the meaning "lines that **end with** our expression".

Note: because '$' and '^' often have special meanings to the command shell itself, we use single quotes to delimit our expression. Since most often you'll want to use Regular Expressions rather than fixed strings, it's a good practice to adopt the usage of delimiting the expression by single quotes whenever you use it.

Example III: inverting grep (using –v)

```
Kraken$ grep –v sh$ /etc/passwd
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
```

Notice none of the output lines end with "sh".

Useful UNIX Filters

- The aunts of all filters: FGREP and EGREP
- Faster (fgrep) or Extended (egrep) versions of GREP
- Generally used by 'advanced' users.

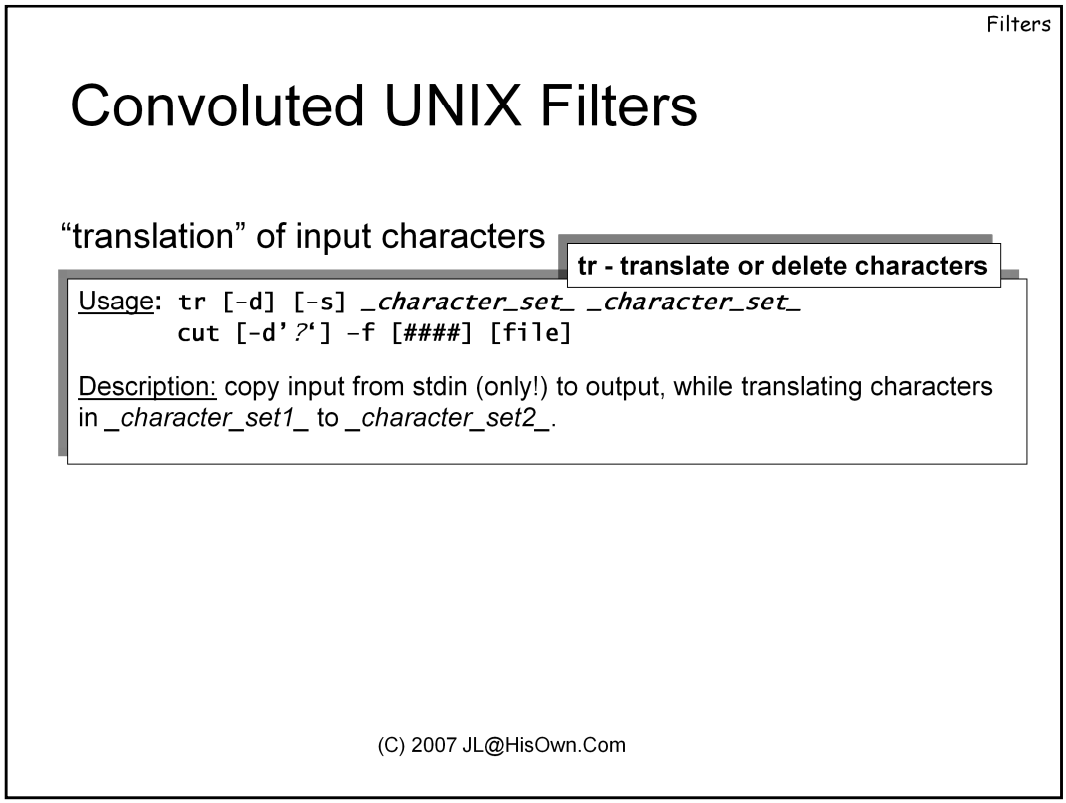
Most UNIX versions support two additional variants of grep – and, in fact, the 'grep' you use is likely to be aliased to one of the two:

fgrep – is a faster version of grep, that supports fixed patterns rather than regular expressions. If all you're looking for is specific text, rather than a variable RegExp, use this grep instead.

egrep – is an extended version of grep, that supports even more regular expressions. Its RegExp engine enables the use of additional characters such as the (|) combination.

Example: Find lines beginning with the word "root" OR the word "adm": Normal grep can't do that.

```
Kraken$ egrep '^(root|adm)' /etc/passwd
root:x:0:0:root:/root:/bin/bash
adm:x:3:4:adm:/var/adm:/sbin/nologin
```



'tr' is a mnemonic for 'translate'. This command is a filter, copying its standard input to standard output, much like 'cat'. But it has further functionality – given two string (character sequence) arguments, source and target, it will replace each character it encounters in its input also found in the source string, with the corresponding character in the same position in the target string.

Is that confusing? Well, as the following will show, this functionality can be used in a variety of ways:

Example I: Uppercase a file

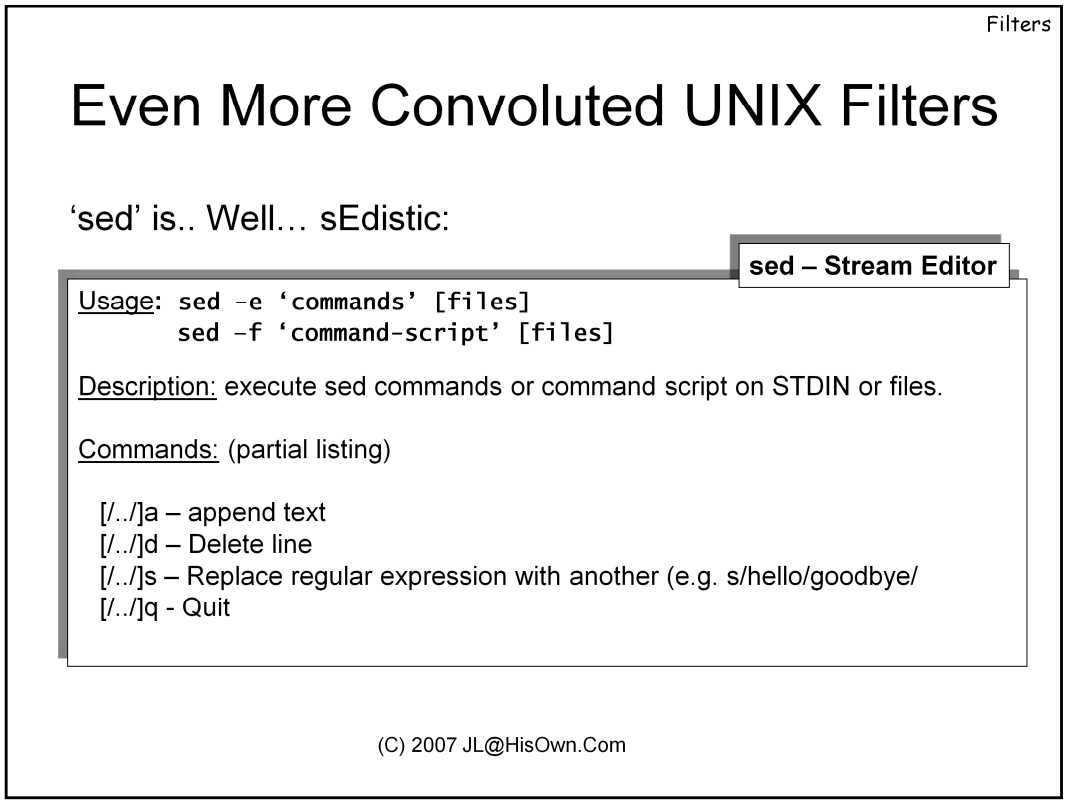
```
Kraken$ tr '[a-z]' '[A-Z]' < /etc/passwd
ROOT:X:0:0:ROOT:/ROOT:/BIN/BASH
BIN:X:1:1:BIN:/BIN:/SBIN/NOLOGIN
DAEMON:X:2:2:DAEMON:/SBIN:/SBIN/NOLOGIN
ADM:X:3:4:ADM:/VAR/ADM:/SBIN/NOLOGIN
.. (output truncated)..
```

Example II: Get rid of characters you really don't like:

```
Kraken$ echo 'xXxyYyxXx' | tr -d 'x'
XyYyX
```

Example III: Squish repeating characters (like space or tab) (***but… why is this useful?***)

```
Kraken$ ls –l
-rw-r--r--  1 johnny users     3595 Sep 19  2006 test.log
Kraken$ ls–l | tr–s ' '
-rw-r--r-- 1 johnny users  3595 Sep 19  2006 test.log
```



'tr' is powerful, but limited: it can only replace specific characters, and has no context determination – It cannot replace characters in specific words, or the words themselves.

This is where sed – the **s**tream **ed**itor – comes into the picture. This is a classic filter: operating from its standard input to its standard output, while processing directives according to regular-expressions.

The full syntax of sed is far, far too complex for our scope. O'Reilly has an entire book devoted to SED and its accomplice, AWK (seriously, these ARE real names of UNIX commands). The following examples, however, show some common uses of this little known utility.

Filters

1. Redirection

Run the 'ls' command with a nonexistent file argument as well as a real one, and capture first its output, then its error stream, into two separate files. Use 'cat' to make sure the files indeed have the proper output. Is everything as you expect?

Now try to capture both output and errors to same file (say, /tmp/redirect). What command would you use? And are the results as expected? If not, explain why?

In order to capture both streams, but not to have one destroy the other, we use a special notation of "2>&1", which essentially tells the shell to "redirect stderr(2) to the same place as it previously did stdout(1)". Try this and make sure this works:

2. Clobbering

Repeat the 'ls' redirection using the ">" operator several times. Each time you try the command, what happens to the file contents? What is the potential issue with this?

Now type: setopt noclobber (zsh), set noclobber (bash), and repeat the previous step. What happens? Does the 'ls' command run? Explain.

Filters (cont)

2. Creative Filter Usage

Explain what the following commands do:

a) head -5 /etc/passwd | tail -1

_____

_____

How would you achieve the same result, with the same commands, but with different arguments?

_____

b) ls –l /tmp | wc –l

_____

c) ls –l /bin | grep '^…x'

_____

d) cut –d':' –f1 /etc/passwd

_____

Filters (cont)

3. xargs

This exercise introduces a new command, called xargs.
Try the following: ‘echo a b c | xargs echo’ and ‘ls | xargs echo’. What is the effect?

_____

_____

Now try: ‘find / -name “/bin/s*” –print | xargs ls

_____

_____

As opposed to find / -name “/bin/s*” –exec ls \{ \};
What is the (barely noticeable) difference?

_____

_____

Filters (cont)

4. Sort, Cut, and friends..

Working with the password file, what are the commands to sort it by…

a) Username (1st column)

_____

b) User Id (3rd column)

_____

c) Full name (description – 5th column)?

_____

Use the "Cut" and sort commands on the password file to create a new file that will contain the columns of full name, username and userid only, in said order.

_____

_____

Use a combination of commands to print out the usernames (only) of all users that are equivalent to the root user (i.e. have a UID of "0")

_____

_____

Use a combination of commands to print out the name of the file last accessed in a given directory

_____

Use a combination of command to print out the name of the largest file in a given directory

_____

Filters (cont)

5. Unleash the power of grep

This exercise shows grep's versatility. Try the following regular expression on any XML or HTML file: `'<\(.*\).*>.*</\1>'`
What is the effect? Explain?

_____

_____

Devise a regular expression to match tags with no child elements.

_____

Advanced: Devise a regular expression to find broken tags

_____

Basic UNIX Scripting

UNIX Shell Scripts

- UNIX Shells are more than mere command interpreters
- Shells support a rich programming language:
  - Users can create their own login scripts
  - Frequent tasks can be automated
  - Existing commands may be grouped and extended
- Shell syntax falls into one of two categories:
  - Bourne: ksh/sh/bash/zsh
  - C: csh/tcsh

Using Variables

- Shells support assignments of arbitrary variables
  - Set variable_name=variable_value to assign
  - Access value by prepending a "$" to variable name
- Variables made be made available to programs
  - Bourne: export C: setenv
- Common practice: Assign command output to variables
  - May be stored for future reference or decision making

Flow Control in shells

- Shells support Decision Making:

```
If expression; then
    ….
else
    …
fi
```

- And looping:

```
for var _list_;
   do


done
```

Commands as boolean operators

- UNIX Commands return an implicit “return code”
  - 0 denotes command success
  - >0 denotes some error condition (depends on command)
- Return value may be collected by:
  - Using the $? Variable
  - Placing the command in an ‘if’ statement
- Commands may be run dependently:
  - cmd1 && cmd2 – Run cmd2 only if 1 succeeds
  - cmd1 || cmd2 – Run cmd2 only if 1 fails

Advanced Scripting

- UNIX supports a host of scripting languages natively:
  - AWK/GAWK/NAWK: A line-editor based scripting language
  - Perl: A full scripting language now incorporated in most UNIX
  - Tcl/Tk: A powerful language with X “ToolKit” support

Basic Scripting

1. Creating your own login script.

Create a simple shell script. Have the script execute a welcome message, set your aliases, and an additional message if it detects the date today is a Wednesday. (hint: use "date" – and consult the man for the right argument).

_____

_____

_____

_____

To enable the login script, copy it into your shell's personal initialization file ***(.zshrc, .kshrc or .bashrc)***

2. Creating a custom command

a) Create a shell script to print out details on a given user in the system. The script will print out the time of last login (by using the "last" command"), as well as the user information (from /etc/passwd) in a more hospitable format.

i) Accept a command line argument by using "$1" in the script. This will be the username

_____

_____

ii) Print an error on too many arguments, by checking the value of $#.

_____

_____

b) Make the script executable (chmod…)

_____

c) Try the script! How would you invoke your script from the command line?

_____

Appendix

UNIX "Cheat Sheet"

Copy files:

cp ***src dst***

src = source file
dst = destination file, or directory (may be . , for current)

Copy a directory tree

cp –pR ***src dst***

src = source directory
dst = destination directory (may be . , for current)
-p = preserve permissions (otherwise you will become owner of these files)
-R = recurse (into subdirectories)

Delete files:

rm –i ***file1 file2***

file1, file2.. = files to delete
-i = interactively. rm nags and asks "are you sure". Safe for beginners

Delete a subtree (deltree):

rm –fR ***dir***

dir = directory to obliterate out of existence
-f = force. rm won't ask nagging questions (but proceed at your own risk)
-R = recurse into subdirectories. This is required since rm cannot remove a directory otherwise.

Move/Rename:

mv ***file_or_dir1 …. target***

file_or_dir1 = File or directory to move/rename
target = new name (for a file/dir) or target directory (for multiple files)
Optional:
-f/-i = force/interactive. mv won't/will ask questions

UNIX "Cheat Sheet" (cont.)

Find a file in the local file system:

find ***where criterion action***

where **=** one or more directories to search in (usually "/")

criterion **=** for search. e.g.

-name foo : Find file named foo

-inum 123: Find filename(s) of inode 123

-size -5000: Find files of up to 5000 bytes

-newer file1: Find files created after file1 was.

action **= -print** : echo the filename (default)

-exec cmd {}\; : execute command on file ({}\; is a placeholder. MUST use \;, no space!)

Display Directory tree (with sizes):

du –k ***dir***

dir **=** directory tree to show. May be omitted, if from current directory (.).

-k **=** display sizes in kb, and not 512-byte blocks.

-s = show summary only (if interested in size, not tree)

Show filesystems, with disk usage:

df ***dir***

dir **=** directory whose filesystem is to show. May be omitted, for all filesystems.

Appendix: (basic) Regular Expressions

Egrep RE's

	Meaning
\	Quote the next metacharacter
^	Match the beginning of the line
.	Match any character (except newline)
$	Match the end of the line (or before newline at the end)
|	Alternation
()	Grouping
[]	Character class

Quantifiers

	Meaning
*	Match preceding 0 or more times
^	Match preceding at least once
?	Match preceding at most once
{n}	Match preceding exactly n times
|{n,}	Match preceding at least n times
{m,n}	Match preceding m<..<n times

Matches are greedy unless '?' is used following quantifier

…If you liked this course, consider…

Protocols:

Networking Protocols – OSI Layers 2-4:

Focusing on - Ethernet, Wi-Fi, IPv4, IPv6, TCP, UDP and SCTP

Application Protocols – OSI Layers 5-7:

Including - DNS, FTP, SMTP, IMAP/POP3, HTTP and SSL

VoIP:

In depth discussion of H.323, SIP, RTP/RTCP, down to the packet level.

Linux:

Linux Survival and Basic Skills:

Graceful introduction into the wonderful world of Linux for the non-command line oriented user. Basic skills and commands, work in shells, redirection, pipes, filters and scripting

Linux Administration:

Follow up to the Basic course, focusing on advanced subjects such as user administration, software management, network service control, performance monitoring and tuning.

Linux User Mode Programming:

Programming POSIX and UNIX APIs in Linux, including processes, threads, IPC and networking. Linux User experience required

Linux Kernel Programming:

Guided tour of the Linux Kernel, 2.4 and 2.6, focusing on design, architecture, writing device drivers (character, block), performance and network devices

Embedded Linux Kernel Programming:

Similar to the Linux Kernel programming course, but with a strong emphasis on development on non-intel and/or tightly constrained embedded platforms

Windows:

Windows Programming:

Windows Application Development, focusing on Processes, Threads, DLLs, Memory Management, and Winsock

Windows Networking Internals:

Detailed discussion of Networking Protocols: NetBIOS/SMB, CIFS, DCE/RPC, Kerberos, NTLM, and networking architecture

Security:

Cryptography:

From Basics to implementations in 5 days: foundations, Symmetric Algorithms, Asymmetric Algorithms, Hashes, and protocols. Design, Logic and implementation

Application Security

Writing secure code – Dealing with Buffer Overflows, Code, SQL and command Injection, and other bugs… before they become vulnerabilities that hackers can exploit.