CST8177 – Linux II

review
Todd Kelley
kelleyt@algonquincollege.com

Final Exam

- CST8177 Linux Operating Systems II
- Mon 22-Apr-13 12:00 14:00 CA105A,B,C

Today's Topics

- assignment09 questions
- assignment10 questions
- a bit more sshd
- a bit more httpd
- review

Examples

- SSH
  - configure port
  - configure logging
- HTTP
  - install it
  - chkconfig it
  - start it
  - firewalling
  - play with default document
  - configuration file to change its behaviors

Course Objectives

- To increase your command line skills
- To add to your knowledge of Linux tools
- To learn basic system administration
- To learn how to design, write, and debug a script
- To provide the required background for the successor courses in later semesters

Brief Course Outline

1. **Control system processes**

the kernel process table; the boot process; log system services; user processes; runlevel tools; task scheduling

2. **Control user access to system resources**

user and group accounts; a password policy; file permissions

3. **Setup and maintain file systems**

volumes; single and multiple file systems; file system integrity

Brief Course Outline

4. **Automate administrative tasks using scripting**
operating system interface; process automation bash scripts

5. Other automation tools (<u>time permitting</u>)
stream editor (**sed**) and **awk**

In summary:

Practical –			
	Lab assignments	25%	
	Online Quizzes	10%	
	Total-P		35%
Theory –			
	Mid-term exam 1	10%	
	Mid-term exam 2	15%	
	Final exam (2 hours)	40%	
	Total-T		65%
	Grand Total		100%

Commands, programs, scripts, etc.

Command

A directive to the shell typed at the prompt. It could be a utility, a program, a built-in, or a shell script.

Program

A file containing a sequence of executable instructions. Note that it's not always a binary file but can be text (that is, a script).

Script

A file containing a sequence of text statements to be processed by an interpreter like bash**, Perl, etc.**

Every program or script has a stdin, stdout**, and** stderr **by default, but they may not always be used.**

Filter

A program that takes its input from stdin **and send its output to** stdout**. It is often used to transform a stream of data through a series of pipes.**

Scripts are often written as filters.

Utility

A program/script or set of programs/scripts that provides a service to a user. (ls, grep, sort, uniq, many many more)

Built-in

A command that is built into the shell. That is, it is not a program or script as defined above. It also does not require a new process, unlike those above.

History

A list of previous shell commands that can be recalled, edited if desired, and re-executed.

Token

The smallest unit of parsing; often a word delimited by white space (blanks or spaces, tabs and newlines) or other punctuation (quotes and other special characters).

stdin

The standard input file; the keyboard; the file at offset 0 in the file table.

stdout

The standard output file; the terminal screen; offset 1 in the file table.

stderr

The standard error file; usually the terminal screen; offset 2 in the file table.

Standard I/O **(Numbered 0, 1, and 2, in order)**

stdin**,** stdout**, and** stderr

Pipe

Connects the stdout **of one program to the** stdin **of the next; the "|" (pipe, or vertical bar) symbol.**

A command line that involves this is called a pipeline

Redirect

To use a shell service that replaces stdin**,** stdout**, or** stderr **with a regular named file.**

Process

http://teaching.idallen.com/cst8207/12f/notes/770_processes_and_jobs.html

- **A process is what a script or program is called while it's being executed. Some processes (called daemons) never end, as they provide a service to many users, such as crontab services from** crond**.**
- **Other processes are run by you, the user, from commands you enter at the prompt. These usually run in the foreground, using the screen and keyboard for their standard I/O. You can run them in the background instead, if you wish.**
- **Each process has a PID (or pid, the process identifier), and a parent process with its own pid, known to the child as a ppid (parent pid). You can look at the running processes with the** ps **command or examine the family relationships with** pstree**.**

Some history examples

- **To list the history:**

```
System prompt> history | less
```

- **To repeat the last command entered:**

```
System prompt> !!
```

- **To repeat the last ls command:**

```
System prompt> !ls
```

- **To repeat the command from prompt number 3:**

```
System prompt> !3
```

- **To scroll up and down the list:**

  Use arrow keys

- **To edit the command:**

  Scroll to the command and edit in place

Redirection

- **Three file descriptors are open and available immediately upon shell startup:** stdin, stdout, stderr
- **These can be overridden by various redirection operators**
- **Following is a list of most of these operators (there are a few others that we will not often use; see** man bash **for details)**
- **Multiple redirection operators are processed from left to right:** redir-1 redir-2 **may not be the same as** redir-2 redir-1
- **If no number is present with** > **or** <, 0 (stdin) **is assumed for** < **and** 1 (stdout) **for** >**; to work with** 2 (stderr) **it must be specified, like** 2>

Operator	Behaviour
	Individual streams
< filename	**Redirects** stdin **from** filename
> filename	**Redirects** stdout **to** filename
>> filename	**Appends** stdout **onto** filename
2> filename	**Redirects** stderr **to** filename
2>> filename	**Appends** stderr **onto** filename
	Combined streams
&> filename	**Redirects both** stdout **and** stderr **to** filename
>& filename	**Same as** &>**, but do not use**
&>> filename	**Appends both** stdout **and** stderr **onto** filename
>>& filename	**Not valid; produces an error**

Operator	Behaviour
	Merged streams
2>&1	**Redirects** stderr **to the same place as** stdout**, which, if redirected, must already be redirected**
1>&2	**Redirects** stdout **to the same place as** stderr**, which, if redirected, must already be redirected**
	Special stdin processing ("here" files), **mainly for use within scripts**
<< <u>string</u>	**Read** stdin **using** <u>string</u> **as the end-of-file indicator**
<<- <u>string</u>	**Same as <<, but remove leading** TAB **characters**
<<< <u>string</u>	**Read** <u>string</u> **into** stdin

Command aliases

- **To create an alias (no spaces after alias name)**

```
alias ll="ls -l"
```

- **To list all aliases**

```
alias   or    alias | less
```

- **To delete an alias**

```
unalias ll
```

- **Command aliases are normally placed in your** ~/.bashrc **file (first, <u>make a back-up copy</u>; then use** vi **to edit the file)**
- **If you need something more complex than a simple alias (they have no arguments or options), then write a bash function script.**

Filename Globbing and other Metacharacters

Metacharacter	Behaviour
\	**Escape; use next char literally**
&	**Run process in the background**
;	**Separate multiple commands**
$xxx	**Substitute variable** xxx
?	**Match any single character**
*	**Match zero or more characters**
[abc]	**Match any one char from list**
[!abc]	**Match any one char** <u>not</u> **in list**
(cmd)	**Run command in a subshell**
{cmd}	**Run in the current shell**

Simple Quoting

- **No quoting:**

```
System Prompt$ echo $SHELL
/bin/bash
```

- **Double quote:** "

```
System Prompt$ echo "$SHELL"
/bin/bash
```

- **Single quote:** '

```
System Prompt$ echo '$SHELL'
$SHELL
```

<u>Observations:</u>

Double quotes allow variable substitution;

Single quotes do not allow for substitution.

Escape and Quoting

- **Escape alone:**

```
Prompt$ echo \$SHELL
$SHELL
```

- **Escape inside double quotes:**

```
Prompt$ echo "\$SHELL"
$SHELL
```

- **Escape inside single quotes:**

```
Prompt$ echo '\$SHELL'
\$SHELL
```

Observations:

Escape leaves the next char unchanged;

Double quotes obey escape (process it);

Single quotes don't process it (treat literally)

Filespecs and Quoting

```
System Prompt$ ls
a  b  c
System Prompt$ echo *
a b c
System Prompt$ echo "*"
*
System Prompt$ echo '*'
*
System Prompt$ echo \*
*
```

<u>Observation</u>:

Everything prevents file globs

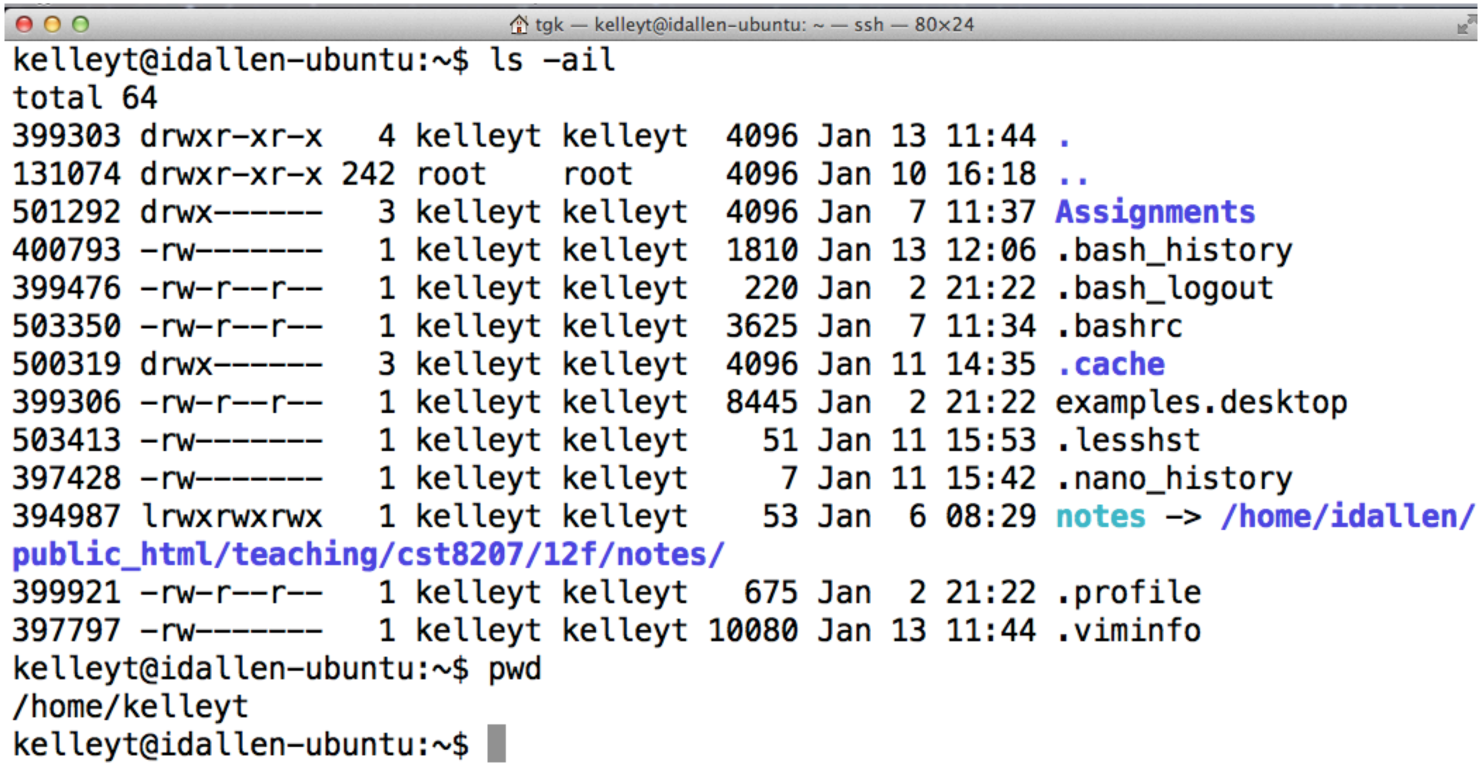
Backquotes and Quoting

```
System Prompt$ echo $(ls)   # alternate
a b c
System Prompt$ echo `ls`       #   forms
a b c
System Prompt$ echo "`ls`"
a
b
c
System Prompt$ echo '`ls`'
`ls`
```

Observations:

Single quotes prevent command processing

File Permissions



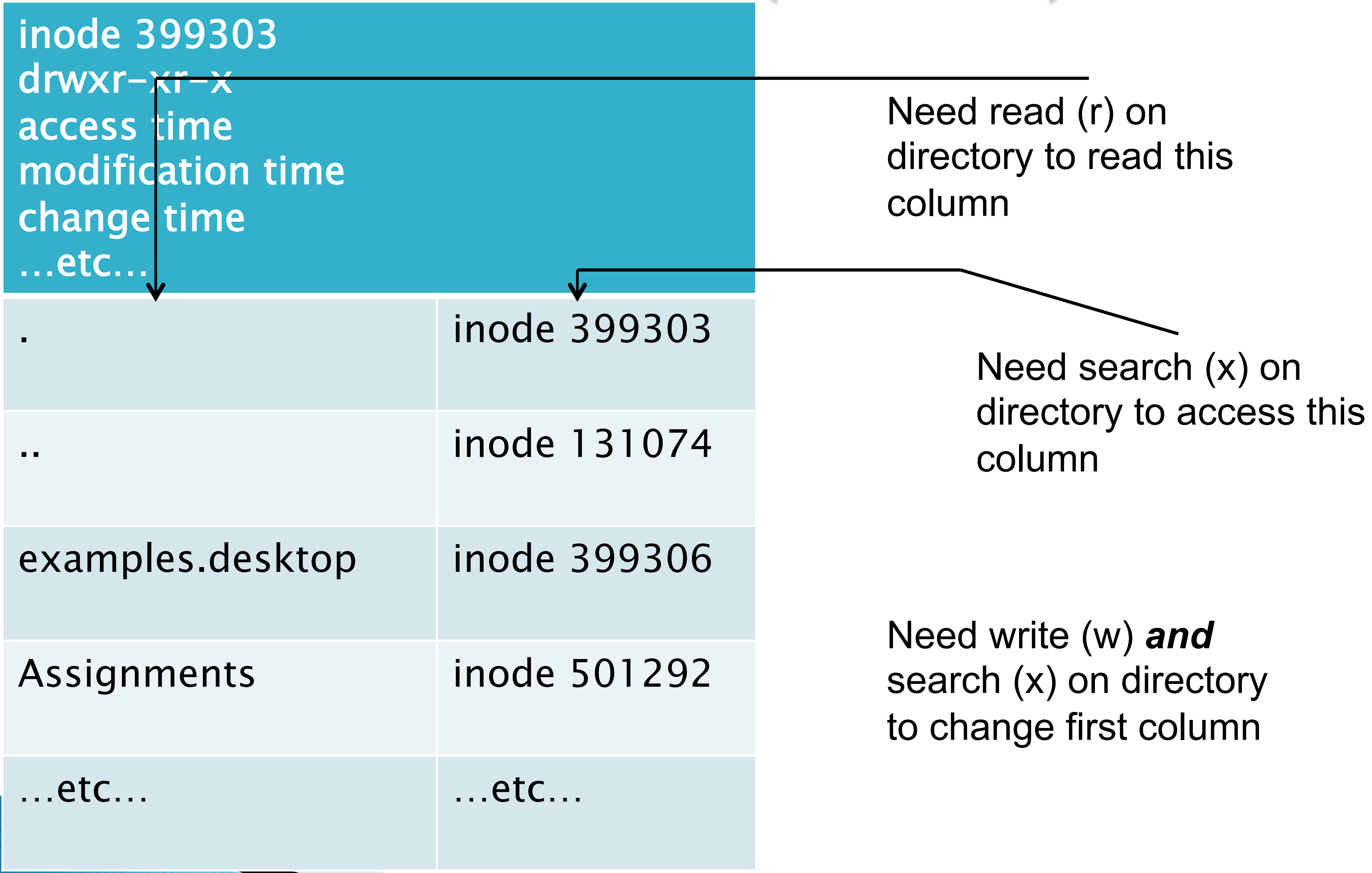
```
kelleyt@idallen-ubuntu:~$ ls -ail
total 64
399303 drwxr-xr-x   4 kelleyt kelleyt  4096 Jan 13 11:44 .
131074 drwxr-xr-x 242 root    root     4096 Jan 10 16:18 ..
501292 drwx------   3 kelleyt kelleyt  4096 Jan  7 11:37 Assignments
400793 -rw-------   1 kelleyt kelleyt  1810 Jan 13 12:06 .bash_history
399476 -rw-r--r--   1 kelleyt kelleyt   220 Jan  2 21:22 .bash_logout
503350 -rw-r--r--   1 kelleyt kelleyt  3625 Jan  7 11:34 .bashrc
500319 drwx------   3 kelleyt kelleyt  4096 Jan 11 14:35 .cache
399306 -rw-r--r--   1 kelleyt kelleyt  8445 Jan  2 21:22 examples.desktop
503413 -rw-------   1 kelleyt kelleyt    51 Jan 11 15:53 .lesshst
397428 -rw-------   1 kelleyt kelleyt     7 Jan 11 15:42 .nano_history
394987 lrwxrwxrwx   1 kelleyt kelleyt    53 Jan  6 08:29 notes -> /home/idallen/
public_html/teaching/cst8207/12f/notes/
399921 -rw-r--r--   1 kelleyt kelleyt   675 Jan  2 21:22 .profile
397797 -rw-------   1 kelleyt kelleyt 10080 Jan 13 11:44 .viminfo
kelleyt@idallen-ubuntu:~$ pwd
/home/kelleyt
kelleyt@idallen-ubuntu:~$
```

Typical directory and file

inode 399303 drwxr-xr-x access time modification time change time …etc…	
.	inode 399303
..	inode 131074
examples.desktop	inode 399306
Assignments	inode 501292
…etc…	…etc…

inode 399306 -rw-r--r-- access time modification time change time …etc…
data blocks for the file there is no filename here the filename(s) (at least one) are stored in directories

File Permissions (cont'd)



inode 399303
drwxr-xr-x
access time
modification time
change time
…etc…

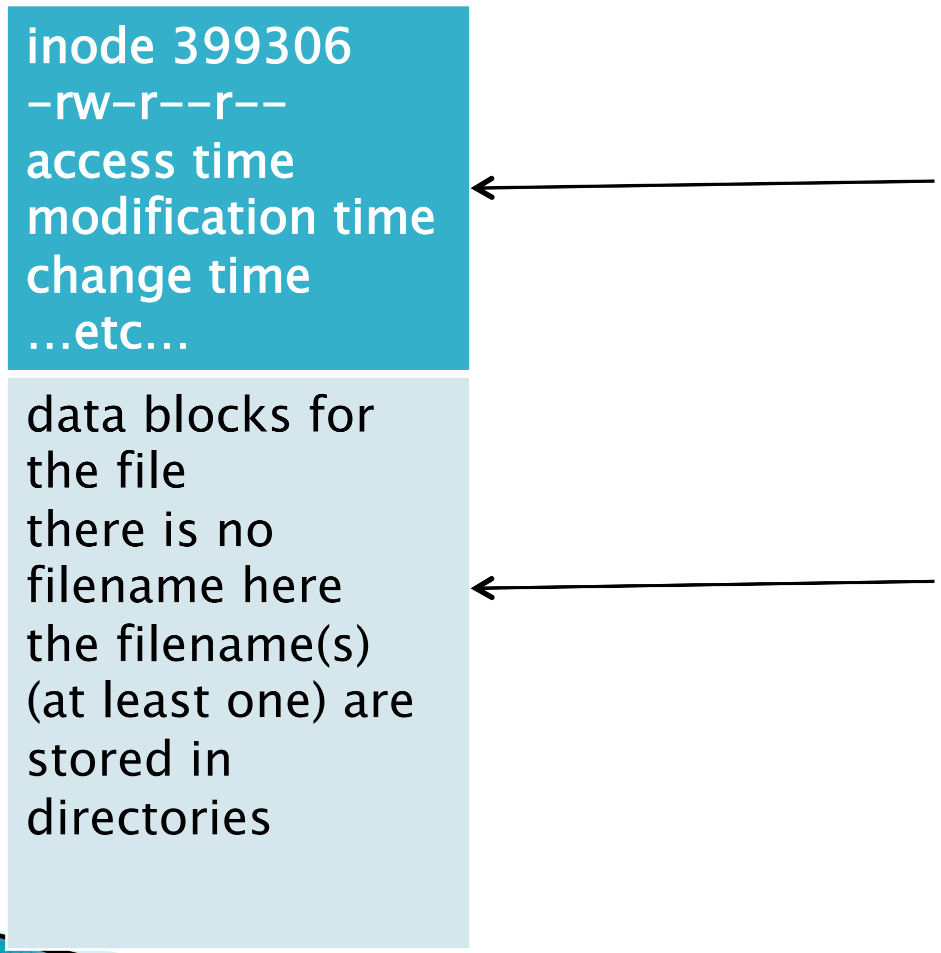
.	inode 399303
..	inode 131074
examples.desktop	inode 399306
Assignments	inode 501292
…etc…	…etc…

Need read (r) on directory to read this column

Need search (x) on directory to access this column

Need write (w) ***and*** search (x) on directory to change first column

File Permissions (cont'd)



inode 399306
-rw-r--r--
access time
modification time
change time
…etc…

data blocks for the file
there is no filename here
the filename(s) (at least one) are stored in directories

Need read (r) and search (x) on ***directory this file is in*** to access this info on the file's inode

Need read (r) / write (w) / execute (x) on ***file*** to read / write / execute this file (contents)

Extending Unix

- create a command with basic scripting
  - put “#!/bin/sh –u” at very beginning of file
  - put commands in file
  - make file executable

- put the file in a directory that is in $PATH
- http://teaching.idallen.ca/cst8207/12f/notes/400_search_path.html

- Not a good idea to put “.” in PATH
- Security implications of putting “current directory” , “.” in path
- PATH=.:$PATH
- demonstration of how the bad guy can arrange for you to inadvertently run their malicious commands as you

Configuring Bash Behavior

- When we customize our shell behavior by
  - setting environment variables (for example, `export PATH=/bin:/usr/bin:/sbin`)
  - setting aliases (for example `alias ll="ls -l"`)
  - setting shell options (for example, `shopt -s failglob` or `shopt -s dotglob`)
  - setting shell options (for example, `set -o noclobber`)

  we make these customizations permanent using bash startup files

Bash Startup Files

- http://teaching.idallen.com/cst8207/12f/notes/210_startup_files.html
- `~/.bash_profile` is sourced by your login shell when you log in
  - the things we set up here are done only once when we log in
  - `export`-ed variables here are inherited by subshells
  - we `source ~/.bashrc` here because login shells do not source it
  - some GUI's log you in without sourcing ~/.bash_profile
- `~/.bashrc` is sourced by each non-login subshell, interactive or not
  - here we set up things that are not inherited by the login shell
  - inside this file, at the top, we check whether it's an interactive or non-interactive shell:

    ```
    [ -z "$PS1" ] && return
    ```
  - we set aliases in this file
  - we set options configured with `shopt` and `set` in this file

.bashrc

```
[ -z "$PS1" ] && return
if [ "${_FIRST_SHELL-}" = "" ] ; then
    export _FIRST_SHELL=$$
    PATH=$PATH:$HOME/bin
    # here we put things that
    # should be done once
fi
# here we put things that need to be
# done for every interactive shell
```

.bash_profile

Contains just one line:

```
[ -f $HOME/.bashrc ] && . $HOME/.bashrc
```

Or equivalently, these three lines instead

```
if [ -f $HOME/.bashrc ]; then
    . $HOME/.bashrc
fi
```

awk (cont'd)

- more generally, we have

`pattern{action}`

- awk reads its input line by line, and for each line that matches `pattern`, the `action` is taken
- If no pattern is specified, then every line matches
- if no action is specified, the default action is `print` (so `awk /this/` is like `grep this`)

awk (cont'd)

- `BEGIN` is a special pattern that matches just before the first actual input line
- `END` is a special pattern that matches just after the last actual input line
- `$0` denotes the whole input line
- `$1` denotes the first field in the input line
- `$2` denotes the second field in the input line, and so on
- `NF` denotes the number of fields
- `FS` denotes the field separator (default whitespace)

awk (cont'd)

- two main ways to set the input field separator
- as an argument on the command line

```
awk -F: '/tgk/{print $7}' /etc/passwd
```

  - this would print field 7, the user's shell, for any password record that contains `tgk`
- Or, we could set the FS variable in a BEGIN action

```
awk 'BEGIN{FS=":"}/tgk/{print $7}' /etc/passwd
```

  - notice that this uses two `pattern{action}` pairs

more awk examples

- for all lines of output from wc, print the first field

```
wc /etc/passwd | awk '{print $1}'
```

- for all lines in the /etc/passwd file, print the number of fields

```
awk -F: '{print NF}' /etc/passwd
```

- for all lines in the /etc/passwd file, print the last field – note difference between NF above and $NF here

```
awk -F: '{print $NF}' /etc/passwd
```

New Commands: stty

- recall the effect of these control characters:
  - `^Z` suspend the current foreground process
  - `^C` terminate the current foreground process
  - `^D` end of file character
  - `^U` kill character to erase the command line
- these are actually properties of the terminal
- they can be set with the stty command
- `stty -a` : print out the current tty settings
- `stty susp ^X` :(that's a caret ^, shift-6 on my keyboard, followed by capital X) means set the susp character to CTRL-X instead of CTRL-Z

stty (cont'd)

- if you accidentally dump the contents of a binary file to your screen, and all the control characters reconfigure your terminal on you, you can reset it to sane values with

```
stty sane
```

Unix Files

- Sobel, Chapter 6
- 160_pathnames.html *Unix/Linux Pathnames (absolute, relative, dot, dot dot)*
- 450_file_system.html *Unix/Linux File System – (correct explanation)*
- 460_links_and_inodes.html *Hard links and Unix file system nodes (inodes)*
- 460_symbolic_links.html *Symbolic Links – Soft Links – Symlinks*
- 500_permissions.html *Unix Modes and Permissions*
- 510_umask.html *Umask and Permissions*

File Types

- Linux recognizes and identifies several file types, which is coded into the first letter of the first field of information about the file:

-	(**dash**)a regular file
b	block device special file
c	character device special file
d	a directory
l	a symbolic (soft) link
p	a named pipe or FIFO
s	socket special filename

File Access Privileges

- In Linux, 3 types of access permissions or privileges can be associated with a file:
  - **read** (r) grants rights to read a file
  - **write** (**w**) grants rights to write to, or change, a file
  - **execute** (**x**) grants rights to execute the file (to run the file as a command)
- All 3 permissions can then be applied to each of 3 types of users:
  - **User:** owner of the file
  - **Group:** group to which user must belong to gain associated rights
  - **Others:** not **User** and not member of **Group** (sometimes called “World” or “Everybody”)

Octal representation of permissions

r w x	Octal Value	Meaning
0 0 0	0	No permission
0 0 1	1	Execute-only permission
0 1 0	2	Write-only permission
0 1 1	3	Write and execute permissions
1 0 0	4	Read-only permission
1 0 1	5	Read and execute permissions
1 1 0	6	Read and write permissions
1 1 1	7	Read, write and execute permissions

Directory Access Privileges

- The same three types of access permissions or privileges are associated with a directory, but with some differences:
  - **read (r)** rights to read the directory
  - **write (w)** rights to create or remove in the directory
  - **execute/search (x)** rights to access the directory meaning, cd into the directory, or access inodes it contains, or "pass through"

All three permissions can then be applied to each of three types of users as before.

– **User**	owner/creator of the file
– **Group**	group to which user must belong
– **Others**	everyone else (Rest-of-world)

Linux File Permissions

- Three special access bits. These can be combined as needed.
- <u>SUID - Set User ID bit</u>
  - When this bit is set on a file, the effective User ID of a process resulting from executing the file is that of the owner of the file, rather than the user that executed the file
  - For example, check the long listing of /usr/bin/passwd – the SUID bit makes this program run as root even when invoked by a regular user – allowing regular users to change their own password

chmod 4xxx file-list

chmod u+s file-list

Linux File Permissions

- SGID - Set Group ID bit
  - Similar to SUID, except an executable file with this bit set will run with effective Group ID of the owner of the file instead of the user who executed the file.

```
chmod 2xxx file-list
chmod g+s file-list
```

Linux File Permissions

- sticky bit (restricted deletion flag)
  - The sticky bit on a directory prevents unprivileged users from removing or renaming a file in the directory unless they are the owner of the file or the directory
  - for example, **/tmp** is a world-writeable directory where all users need to create files, but only the owner of a file should be able to delete it.
  - without the sticky bit, hostile users could remove all files in /tmp; whereas with the sticky bit, they can remove only their own files.

chmod 1xxx dir-list

chmod +t dir-list

Linux File Permissions

- The permissions a user will have is determined in this way:
  - If the user is the owner of the file or directory, then the **user** rights are used.
  - If the user is not the owner but is a member of the group owning the file or directory, then the **group** rights are used.
  - If the user is neither the owner nor a part of the group owning the file, then the **other** rights are used.
- NOTE: It is possible to give the “world” more permissions that the owner of the file. For example, the unusual permissions `-r--rw-rw-` would prevent only the owner from changing the file – all others could change it!

umask

- The permissions assigned to newly created files or directories are determined by the **umask** value of your shell.
- Commands:
  - **umask** - display current umask
  - **umask xyz** - sets new umask to an octal value **xyz**
- permissions on a newly created file or directory are calculated as follows:
  - start with a "default" of 777 for a directory or 666 for a file
  - for any 1 in the binary representation of the umask, change the corresponding bit to 0 in the binary representation of the default
- umask is a reverse mask: the binary representation tells you what bits in the 777 or 666 default will be 0 in the permissions of the newly created file or directory

umask examples (Files)

- if umask is 022
  - binary umask representation: 000010010 = 022
  - default file permissions 666: 110110110
  - permissions on new file: 110100100 = 644
- if umask is 002
  - binary umask representation: 000000010 = 002
  - default file permissions 666: 110110110
  - permissions on new file: 110110100 = 664
- if umask is 003
  - binary umask representation: 000000011 = 003
  - default file permissions 666: 110110110
  - permissions on new file: 110110100 = 664

umask examples (Files, cont'd)

- notice that for files, a umask of 003 ends up doing the same thing as a umask of 002
- Why?

umask examples (Directories)

- if umask is 022
  - binary umask representation: 000010010 = 022
  - default dir permissions 777: 111111111
  - permissions on new dir : 111101101 = 755
- if umask is 002
  - binary umask representation: 000000010 = 002
  - default dir permissions 777: 111111111
  - permissions on new dir : 111111101 = 775
- if umask is 003
  - binary umask representation: 000000011 = 003
  - default dir permissions 777: 111111111
  - permissions on new dir : 111111100 = 774

umask examples (Dirs, cont'd)

- notice that for directories, a umask of 003 gives different results than a umask of 002
- Why?

Linux File Permissions

- It is important for the Linux file system manager to govern permissions and other file attributes for each file and directory, including
  - ownership of files and directories
  - access rights on files and directories
  - The 3 timestamps seen in **stat (man stat)**
- The information is maintained within the file system information (**inodes**) on the hard disk
- This information affects every file system action.

Linux Basic Admin Tools

- **chown owner[:group] files**
  - Change ownership of files and directories (available for **root** only)

Examples:

chown guest:guest file1 dir2

- change ownership of **file1** and **dir2** to user **guest** and group **guest**

chown guest dir2

- change ownership of **dir2** to user **guest** but leave the group the same

chown :guest file1

- change ownership of **file1** to group **guest** but leave the user the same (can also use **chgrp**)

Linux Basic Admin Tools

- **chmod permissions files**
  - Explicitly change file access permissions

Examples:

chmod +x file1

- changes **file1** to have executable rights for user/group/ other, subject to umask

chmod u+r,g-w,o-rw file2

- changes **file2** to add read rights for user, remove write rights for group and remove both read and write rights for others

chmod 550 dir2

- changes **dir2** to have only read and execute rights for user and group but no rights for other

Regular Expressions (again)

- Three kinds of matching
  1. Filename globbing
     - used on shell command line, and shell matches these patterns to filenames that exist
     - used with the `find` command
  2. Regular expressions, used with
     - vi
     - sed
     - awk
     - grep
  3. Extended regular expressions
     - egrep or grep -E (not emphasized in this course)
     - perl regular expressions (not in this course)

Matching Patterns

- filename globbing patterns match existing pathnames in the current filesystem only
- Globbing is used for
  - globbing patterns in command lines
  - patterns used with the find command
  - examples:
    - `ls *.txt`
    - ls ?????.txt
    - ls [ab]*.txt
    - find ~ –name “*.txt”

Regular Expressions

- IMPORTANT: regular expressions use some of the same special characters as filename matching on the previous slide but they mean different things!
- Read under `REGULAR EXPRESSIONS` in the man page for the `grep` command – this tells you what you need to know
- The grep man page is normally available on Unix systems, so you can use it to refresh your memory, even years from now

POSIX character classes

- **[:alnum:] a – z, A – Z,** and **0 – 9**
- **[:alpha:] a – z** and **A – Z**
- **[:cntrl:]** control characters
- **[:digit:]** 0 – 9
- **[:lower:] a – z**
- **[:print:]** visible characters, plus **[:space:]**
- **[:punct:]** Punctuation characters and other symbols
  - !"#$%&'()*+,\-./:;<=>?@[\\\]^_`{|}~
- **[:space:]** White space (space, tab)
- **[:upper:]** A – Z
- **[:xdigit:]** Hex digits: **0 – 9, a – f,** and **A – F**
- **[:graph:]** (**0x21** – **0x7E**) (we won't use)

Testing Regular Expressions

- testing regular expressons with grep on stdin
  - run grep ‘expr’ on the standard input
  - use the single quotes to protect your expr from the shell
  - grep will wait for you to repeatedly enter your test strings (type ^D to finish)
  - grep will print any string that matches your expr, so each matched string will appear twice (once when you type it, and once when grep prints it)
  - unmatched strings will appear only once where you typed them
  - type ^D to finish

Regular Expressions to test

- examples (try these)
  - grep ‘ab’ #any string with **a** followed by **b**
  - grep ‘aa*b’ #one or more **a** followed by **b**
  - grep ‘a..*b’ #**a,** then one or more anything, then **b**
  - grep ‘a.*b’ #**a** then zero or more anything, then **b**
  - grep ‘a.b’ # **a** then exactly one anything, then **b**
  - grep ‘^a’ # **a** must be the first character
  - grep ‘^a.*b$’ # **a** must be first, **b** must be last
- Try other examples: have fun!

Shell scripting

- For the impatient, you can read ahead

http://elearning.algonquincollege.com/coursemat/alleni/idallen/cst8177/13w/notes/000_script_style.html

- From now on, at the top of all our shell scripts, we put

```
#!/bin/sh -u
# UTF-8 (international) script header
PATH=/bin:/usr/bin ; export PATH
umask 022
unset LC_ALL                           # unset the over-ride variable
LC_COLLATE=en_US.utf8 ; export LC_COLLATE # sort by character set
LC_CTYPE=en_US.utf8 ; export LC_CTYPE # handle multi-byte chars
LANG=en_US.utf8 ; export LANG          # legacy version of LC_CTYPE
```

Internationalization (i18n)

- http://teaching.idallen.com/cst8177/13w/notes/000_character_sets.html

- Not all computer users use the same alphabet
- When we write a shell script, we need to ensure that it handles text properly in the presence of i18n
- In the beginning, there was ascii, a 7 bit code of 128 characters
- Now there's Unicode, a table that is meant to assign an integer to every character in the world
- UTF-8 is an implementation of that table, encoding the 7-bit ascii characters in a single byte with high order bit of 0
- The 128 single-byte UTF-8 characters are the same as true ascii bytes (both have a high order bit of 0)
- UTF-8 characters that are not ascii occupy more than one byte
- Locale settings determine how characters are interpreted and treated, whether as ascii or UTF-8, their ordering, and so on

Positional Parameters

- $# holds the number of arguments on the command line, not counting the command itself
- $0 is the name of the script itself
- $1 through $9 are the first nine arguments passed to the script on the command line
- After $9, there's ${10}, ${11}, and so on
- $* and $@ denote all of the arguments
- "$*" is one word with spaces in it
- "$@" produces a list where each argument is a separate word

Sample script

```
#!/bin/sh -u
PATH=/bin:/usr/bin ; export PATH
umask 022
unset LC_ALL                          # unset the over-ride variable
LC_COLLATE=en_US.utf8 ; export LC_COLLATE # sort by character set
LC_CTYPE=en_US.utf8 ; export LC_CTYPE # handle multi-byte chars
LANG=en_US.utf8

echo "The number of arguments is: $#"
echo "The command name is $0"
echo "The arguments are $*"
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "The third argument is: $3"
```

Interacting with the user

- to get input from the user, we can use the `read` builtin
- `read` returns an exit status of 0 if it successfully reads input, or non-zero if it reaches EOF
- `read` with one variable argument reads a line from stdin into the variable
- Example:

```
#!/bin/sh
read aline #script will stop, wait for user
echo "you entered: $aline"
```

Interacting with the user (cont'd)

- Use the -p option to read to supply the user with a prompt
- Example

```
#!/bin/sh -u
read -p "enter your string:" aline
echo "You entered: $aline"
```

Interacting with the user (cont'd)

- `read var1` puts the line the user types into the variable `var1`
- `read var1 var2 var3` puts the first word of what the user types in to `var1`, the second word into `var2`, and the remaining words into `var3`

```
#!/bin/sh -u
read var1 var2 var3
echo "First word: $var1"
echo "Second word: $var2"
echo "Remaining words: $var3"
```

Exit Status

- Each command finishes with an exit status
- The exit status is left in the variable ? ($?)
- A non-zero exit status normally means something went wrong (grep is an exception)
- non-zero means "false"
- A exit status of 0 normally means everything was OK
- 0 means "true"
- grep returns 0 if a match occurred, 1 if not, and 2 if there was an error

If statement

```
if list1; then
    list2;
fi
```

- `list1` is executed, and if its exit status is 0, then `list2` is executed
- a list is a sequence of one or more pipelines, but for now, lets say it's a command

Test program

- A common command to use in the test list of an if statement is the `test` command
- `man test`
- Examples:
- `test -e /etc/passwd`
- `test "this" = "this"`
- `test 0 -eq 0`
- `test 0 -ne 1`
- `test 0 -le 1`

If statement with test

```
if test "$1" = "hello"; then
   echo "First arg is hello"
fi

if test "$2" = "hello"; then
   echo "Second arg is hello"
else
   echo "Second arg is not hello"
fi
```

The program named [

```
Todd-Kelleys-MacBook-Pro:CST8177-13W tgk$ ls -li /bin/test /bin/[
1733533 -r-xr-xr-x  2 root  wheel  43120 27 Jul  2011 /bin/[
1733533 -r-xr-xr-x  2 root  wheel  43120 27 Jul  2011 /bin/test
Todd-Kelleys-MacBook-Pro:CST8177-13W tgk$
```

- notice that `[` is another name for the `test` program:

```
if [ -e /etc/passwd ]; then
    echo "/etc/passwd exists"
fi
```

is the same as

```
if test -e /etc/passwd; then
    echo "/etc/passwd exists"
fi
```

Practicing with [

```
$ [ 0 –eq 0 ]
$ echo $?
0
$ [ "this" = "that" ]
$ echo $?
1
$ [ "this" = "this" ]
echo $?
0
$ ["this" = "this"]                    # forgot the space after [
-bash: [this: command not found
$ [ "this" = "this"]                   # forgot the space before ]
-bash: [: missing ']'
```

For loop

```
for name [ in word... ] ; do list ; done
```

- `name` is a variable name we make up
- `name` is set to each `word...` in turn, and list is exectuted
- if `[ in word... ]` is omitted, the positional parameters are used instead

For loop example

```
for f in hello how are you today; do
   echo "Operating on $f"
done
```

While loop

```
while command; do
   # this code runs over and over
   # until command has
   # non-zero exit status
done
```

While loop example

```
while read -p "enter a word: " word; do
    echo "You entered: $word"
done
```

Until loop

- "opposite" of while

```
until [ "$word" = END ]; do
   read -p "Enter a word:“ word
   echo "You entered $word"
done
```

Basic sed

- we'll use sed to read lines from stdin or a file, and write the modified lines to stdout
- we'll concentrate on the forms
  - `sed 's/this/that/'`
    - replace first instance of `this` with `that`
  - `sed '/^#/s/this/that/'`
    - on lines that begin with `#` replace first instance of `this` with `that`
  - `sed 's/this/that/g'`
    - replace all instances of `this` with `that`
  - `sed -e 's/this/that/' -e 's/what/who/'`
    - replace first instance of `this` with `that` and first instance of `what` with `who`

Sed examples

```
echo this | sed 's/this/that/'
that
echo this and this | sed 's/this/that/'
that and this
echo this and this | sed 's/this/that/g'
that and that
echo this and what | sed -e 's/this/that/' -e 's/what/why/'
that and why
echo this and that | sed -e 's/this/that/' -e 's/that/why/g'
why and why
```

combining tests together

- building complex tests from simple tests
- test1 -a test2
  - both test1 and test2 must be true
- test1 -o test2
  - at least one must be true
- ! test1
  - test1 must be false
- (test1)
  - true if test1 is true

&& means "and"

- Suppose you might qualify for a scholarship:
- Those who qualify are:
  - eight feet tall, and ??
  - born on the moon, and ??
  - algonquin student and ??
- or in other words
  - eight feet tall `&&` ??
  - born on the moon `&&` ??
  - algonquin student `&&` ??
- In which case do we need to find out what ?? is?

|| means "or", opposite of &&

- As soon as we encounter "true", we can stop
- You qualify for a $1000 rebate under the following conditions:
  - born on the moon, or ??
  - algonquin student, or ??
- In the first case, we need to know what the exit status of the ?? is, we need to run the ?? command
- In the second case, we can stop before running the ?? command

&& and || versus -a and -o

- && and || are used with commands that tend to get things done
  - to graduate, you
    - complete first year && complete second year
  - complete first year is a "command" that gets things done: you learn the first-year material
- -a and -o are used in test, and don't do things, just affect the exit status of test
  - you are a rich canadian if
    - you are canadian -a you are rich
  - checking whether or not you're canadian doesn't get things done – but it does establish a truth value

set

- set with arguments but no options sets the positional parameters to the arguments
- $# is the number of arguments
- $1 is the first arg
- $2 is the second arg
- $3 is the third arg
  - etc
- $@ and $* are all args

shift

- shift
- moves all the arguments to the left
- shift n moves all the arguments to the left by n
- shift
  - $# is decreased by 1
  - the pre-shift $1 is lost
  - $1 becomes what was in $2
  - $2 becomes what was in $3
  - $3 becomes what was in $4
    - etc

Debugging shell scripts

- -v option for bash/sh
  - sh -v myscript
  - shell will print each line as its read
  - loop statements are printed once

- -x option for bash/sh
  - sh -x myscript
  - shell will display $PS4 prompt and the expanded command before executing it
  - each loop iteration is shown individually

exit command

- `exit` causes the shell to exit with the exit status of the last command that was run
- `exit N` causes the shell to exit with exit status `N`

case statement

```
case test-string in
    pattern-1)
        command1
        command2
        ;;
    pattern-2)
        command3
        command4
        ;;
    *)
        command5
        ;;
esac
```

case statement continued

- the patterns are globbing patterns matched to the test-string
- So we tend to use the * pattern as a catchall, if all other matches fail, but that's not required
- case statement exit status is the exit status of the last command in the matching block, or 0 if no blocks match

case statement continued

- We can use the vertical bar to specify alternative patterns:

```
case "$character" in
    a|A)
        echo "The character is A"
        ;;
    [bB])
        echo "the character is B"
        ;;
    *)
        echo "The character is not A or B"
        ;;
esac
```

Is stdin a terminal?

- A script can test whether or not standard input is a terminal

```
[ -t 0 ]
```

- What about standard output, and standard error?

The command that does nothing

- Occasionally you'll see a command called :
- `: arguments`
- That command expands its arguments and does nothing with them, resulting in a 0 exit status

Doing integer arithmetic

- examples of using `expr` command:

```
a=`expr 3 + 4`
a=`expr 3 - 4`
a=`expr 3 * 4`
a=`expr 13 / 5`   # integer division: 2
a=`expr 13 % 5`   # remainder: 3
```

- increment the integer in variable `a`

```
a=`expr $a + 1`
```

ps command

- We have already been using the ps command to print out information about processes running on the system
- ps -ef or ps aux piped to grep is common
- there are many options for printing specific info in a specific way: man ps or ps -h
- ps -l # long format
- ps -f versus ps -fw

top command

- top displays some system information, and a list of processes, ordered on a column
- the most important keys are ?, h, and q (according to man page)
- load average: 5min, 10min, 15min
- load average is number of processes running or in uninterruptable state (disk IO, others)
- no exact rule, but if load average is more than 1-1.5 times the number of CPUs, the machine is overloaded in some way and you have a problem (your mileage may vary)

Nice command

- Each process has a priority, which you can control with the nice command
- -20 highest priority, 19 lowest priority
- nice [-n increment] command
- nice -n 10 long_command # 10 is default
- only superuser can specify negative increments
- For processes already running:
  - renice priority -p PID or renice -n increment -p PID

Job Control

- your shell can run several processes for you at once
- we can run commands in the background
  - command &
- we can put a running command in the background
  - ^Z
- what jobs are there?
  - jobs
- resume a stopped job
  - bg %N   # background, where N is a job number
  - fg %N   # foreground

Sending signals: kill command

- When we type ^C when a process is running in the foreground, the process receives a SIGINT signal, which by default would cause a process to terminate.
- SIGINT: ^C (default), similar to SIGTERM
- SIGHUP: terminal has been closed
- SIGTERM: clean up if necessary, then die
- SIGKILL: die right now
- We can send these signals to a process with the kill command

Send a signal to kill a process

- kill –SIGNAL PID #send SIGNAL to process PID
- When system shuts down, it
  - sends all processes a SIGTERM
  - waits a few seconds (5 or 10)
  - sends all processes a SIGKILL
- Why not just wait for the SIGTERM to finish?
- Because SIGTERM can be handled, possibly ignored, it's optional
- SIGKILL cannot be handled – it works unless the process is in an uninterruptible state (maybe disk I/O, NFS)

Configuring your cron job

- full details from man 5 crontab
  - recall that is how we read section 5 of the manual (section 5 of the manual is file formats)
- man crontab will give info about the crontab command (in default section 1 of the manual)
- create a file containing your cron instructions (see next slide), naming that file, say, myuser.crontab
- run the crontab command to submit that file's contents to be your user's crontab file: crontab < myuser.crontab
- alternatively, you can edit your user's live crontab file: crontab -e

crontab format (man 5 crontab)

- All fields must contain a value of some valid kind
- Field are separated by one or more spaces
- Asterisk (*) indicates the entire range

```
# .---------------- minute (0 - 59)
# |   .------------- hour (0 - 23)
# |   |   .---------- day of month (1 - 31)
# |   |   |   .------- month (1 - 12)
# |   |   |   |   .---- day of week (0 – 7, both 0 and 7 are Sunday)
# |   |   |   |   |
  0  6   1  *  *  /home/user/bin/mycommand
  1  6  15  *  *  /home/user/bin/anothercommand > /dev/null 2>&1
```

crontab format (cont'd)

- ranges with dash are allowed: first-last
- * means every value first-last
- lists are allowed: first,second,third
- steps indicated with '/' are allowed after ranges or asterisk:
  - */2 means every second one
  - 1-7/2 means 1,3,5,7

common crontab options

- crontab -l
  - list the contents of your current live crontab file
- crontab -e
  - edit the contents of your current live crontab file
- crontab
  - read the new contents of for your crontab file from stdin
- crontab -r
  - remove your current crontab file

example crontab

- see man 5 crontab for example crontab
- really, see the example: man 5 crontab
- things to watch out for
  - input for your commands (they run without anyone to type input)
  - output of commands (if you don't (re)direct output, the output will be emailed – better if you handle it)
  - error output of commands (same as for output above)
  - summary: it's best if your commands in a crontab are arranged with input and output already handled, not relying on output to be emailed by cron
  - if you want to email, do it explicitly in your command somehow, and test that command before putting it into your crontab

at command

- at command runs a set of commands at a later time
- at command takes a TIME parameter and reads the set of commands from standard input
- example (run commands at 4pm 3 days from now)

  -
    ```
    at 4pm + 3 days
    rm -f /home/usr/foo
    touch /home/usr/newfoo
    ^D
    ```
- other at-related commands: atrm, atq
- for details: man at
- as with cron, you must be aware of how your commands will get their input (if any) and what will happen to their output (if any)

Healthy Multi-User System

- an account has been created for everyone who should have one (the users)
- every user is authorized to read, write, and execute exactly what they should be able to
  - not more
  - not less
- every user can access the resources they need
  - disk space
  - software applications/libraries
  - processes, memory, CPU time
  - resource hogs don't affect the work of other users

Healthy Multi-User System (cont'd)

- Accessible to its users
  - accessible remotely if applicable (ssh)
  - good uptime with reasonable maintenance windows
- Secure from attack
  - inaccessible to unauthorized users (external attack)
  - no unauthorized or stolen access to user accounts
  - resistant to internal attacks
    - users cannot elevate their privileges
    - users don't bring system down without trying
  - prevent cross-user attacks
    - ensure users cannot interfere with each other's
      - confidentiality of files
      - integrity of files
      - availability of files

Regular Maintenance

- backups
- security patches
- monitor and manage disk space
  - find and educate and control "space hogs"
  - add new disk space
  - replace failed disk space
- software installation
- software updates
- system upgrades (preferably not often)
- monitor the system logs for issues

Three types of account

- Root account
  - having a root password is not necessary
  - not having a root password means one less password to manage, one less vulnerability
  - root access is gained by system administrators
- System Administrator
  - configured in sudoers file
  - gain root privileges with `sudo -s`
- Regular User
  - often named according to a pattern
  - this is the kind of account you have on the CLS

Setting up root

- common model is to put sysadmins in sudoers file
- as root, do `visudo`
- put the following line in
  - `youradminname ALL=(ALL) ALL`
  - youradminname: the username you use for admin
  - ALL: from any host
  - (ALL): run commands as any user
  - ALL: run any command
- test that you can become root with `sudo -s`
- put * in root password field in /etc/shadow

User Management

- Create, Modify, and Remove User Accounts
- Create, Populate, Modify, and Remove Groups
- Password Policy
  - strength of passwords
  - how often passwords must be/can be changed
  - how often passwords can be reused (or based on an old password)
- Set and Administer File Permissions
- http://teaching.idallen.com/cst8207/12f/notes/600_users_and_groups.html

creating many new users (cont'd)

- to create one user:

```
useradd -c "Full Name" user001
chmod 750 /home/user001
passwd user001 # and enter passwd by hand
```

creating many new users (cont'd)

- there are various possible strategies for creating many new user accounts
- one possibility:
  - use Linux utilities and/or your own script to create a set of commands for each new user (one-off script):

```
useradd -c "User 1" user001            #create the user
usermod -p u75jjvrue5B92 user001  #assign passwd
chmod 750 /home/user001 || exit 1 #homedir perms
```

- If you were creating 100 users, you'd have 300 commands in your one-off script

creating many users (cont'd)

- another possibility: the "newusers" command
- man newusers
- newusers takes a file containing info about the accounts you want to create
- the input file for creating the accounts is in the same format as the /etc/passwd file:

```
uncle:3uncle4:503:503:Uncle Tom:/home/uncle:/bin/bash
aunt:3aunt4:504:504:Aunt Betty:/home/aunt:/bin/bash
```

Yum: Yellowdog Updater Modified

- http://teaching.idallen.com/cst8207/13w/notes/810_package_management.html
- yum can install software packages for you, retrieving them from a repository over the network
- performs dependency analysis: if the package you want to install depends on another package, it will install that too
- can also query installed packages, remove packages, update packages, etc
- run with root privileges

Yum (cont'd)

- Examples: (see "man yum" for details)
  - yum install ntp
    - install the package "ntp" and its dependencies
  - yum update
    - update all currently installed packages
  - yum update "nt*"   # quote the glob from the shell
    - update all packages that match the glob
  - yum –v repolist
  - yum list installed
  - yum list available
  - yum list   # combination of two above
  - yum search fortune

Yum repository configuration

- we shouldn't need to change these, but if you're curious...
- repository files are in /etc/yum.repos.d
  - CentOS-Base.repo
    - main CentOS repository mirrors
  - CentOS-Media.repo
    - uses the DVD in your drive as a repository

tar command basics

- create an archive of a directory
  - tar cvzf mydirectory.tgz mydirectory
    - c: create an archive
    - v: verbose, print the filenames as their added
    - z: compress the archive
    - f: use the following as the filename for the archive
- extract an archive
  - tar xvzf mydirectory.tgz
    - x: extract an archive
    - z: uncompress the archive

tar command basics (cont'd)

- print listing of an archive without extracting
  - tar tvzf mydirectory.tgz mydirectory
    - t: print a listing
    - v: verbose, like a long listing
    - z: the archive is compressed
    - f: use the following as the filename for the archive
- In each of the above examples
  - exactly one of t, c, or x is mandatory
  - f with an archive name is mandatory
  - z: is mandatory if archive is, or is to be, compressed
  - v: is optional for verbosity

rsync basics: Trailing slash

- be careful with a trailing slash on the source
- a trailing slash on source has special meaning: copy the contents of the directory
- these are the same
  - `rsync -avH /src/foo    /dst/`
  - `rsync -avH /src/foo/  /dst/foo`
- copy contents of src directory to dst directory
  - `rsync -avH /src/ /dst        # /src/* in /dst/`
- copy src directory to dst directory
  - `rsync -avH /src /dst    #end up with /dst/src`

rsync (cont'd)

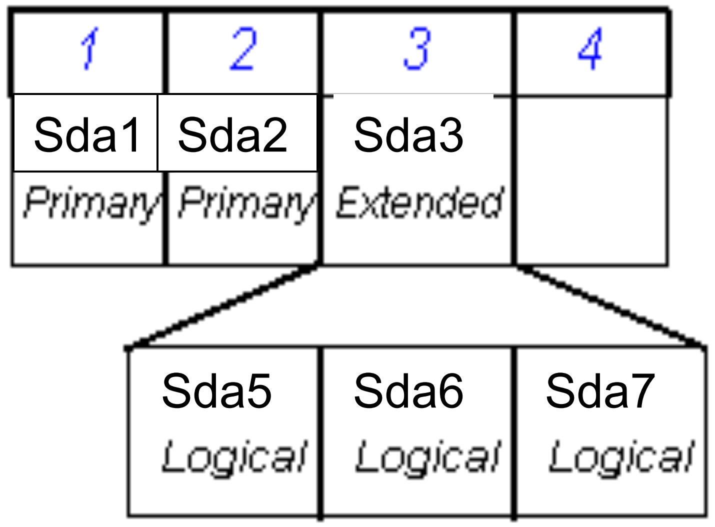
- rsync can copy across the network

```
rsync -avH dir/.   kelleyt@remote.example.com:dir
```

- that will copy/synchronize the local "dir" with the remote "dir" in kelleyt's home dir on the remote machine named “remote.example.com”
- notice the colon in the remote argument
- if you forget the colon, you do a local copy to

a file with '@' in its name

Identifying Partitions

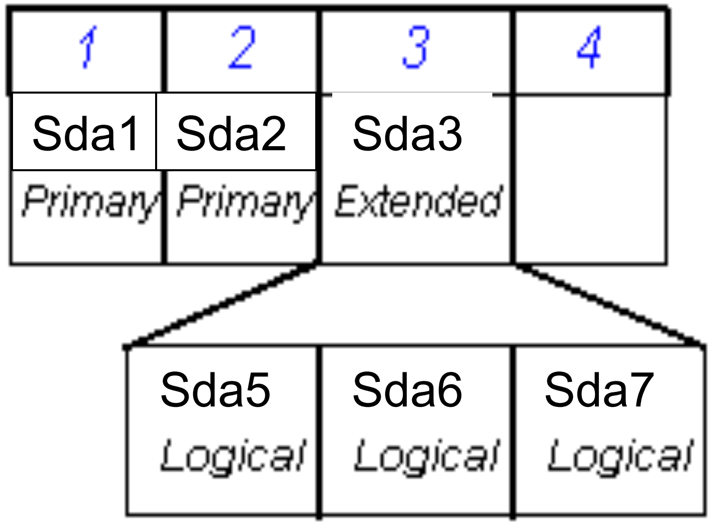


Naming partitions

- sd*x*1 – sd*x*4
  - Primary Partitions recorded in the partition table
- sd*x*5 – sd*x*63
  - Logical partitions

Note: You can have up to 4 primary partitions created in your system, while there can be only one extended partition.

Identifying Partitions



Naming partitions

- sd*x*1 – sd*x*4
  - Primary Partitions recorded in the partition table
- sd*x*5 – sd*x*63
  - Logical partitions

Note: You can have up to 4 primary partitions created in your system, while there can be only one extended partition.

File systems (8207 review)

- http://teaching.idallen.com/cst8207/13w/notes/720_partitions_and_file_systems.html

Linux/Unix mounting

- no drive letters!

/dev/sda2

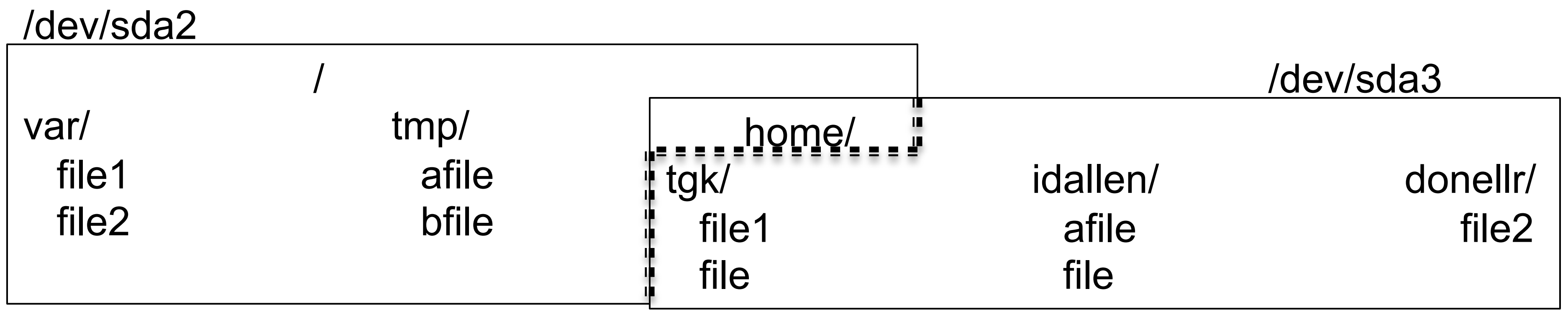
```
                /
var/                  tmp/              home/
  file1                 afile             dir1/
  file2                 bfile                file1
                                             file 2
```

/dev/sda3

```
          /
tgk/                  idallen/            donellr/
  file1                 afile               file2
  file                  file
```

Linux/Unix mounting

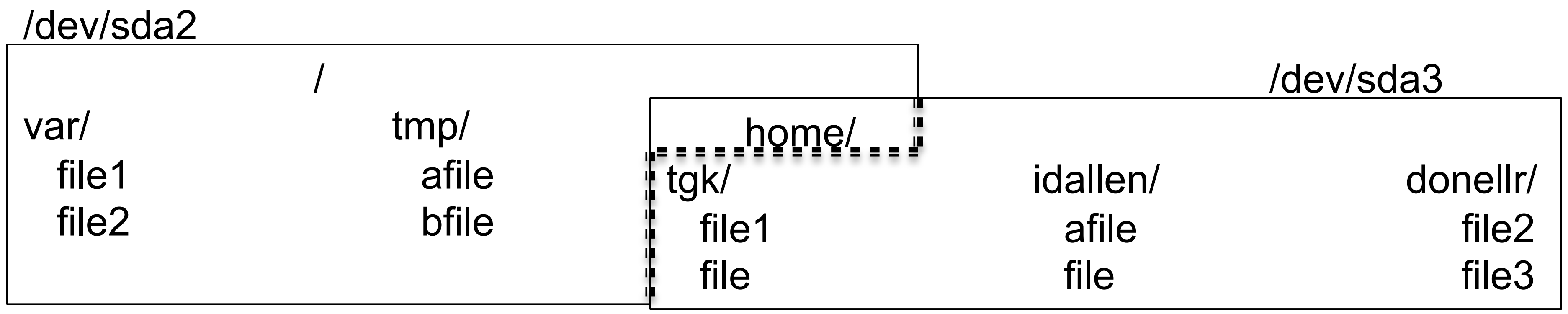
- mount /dev/sda3 /home



- the /home directory name still on /dev/sda2
- the contents of /home are on /dev/sda3
- the previous contents of /home are hidden

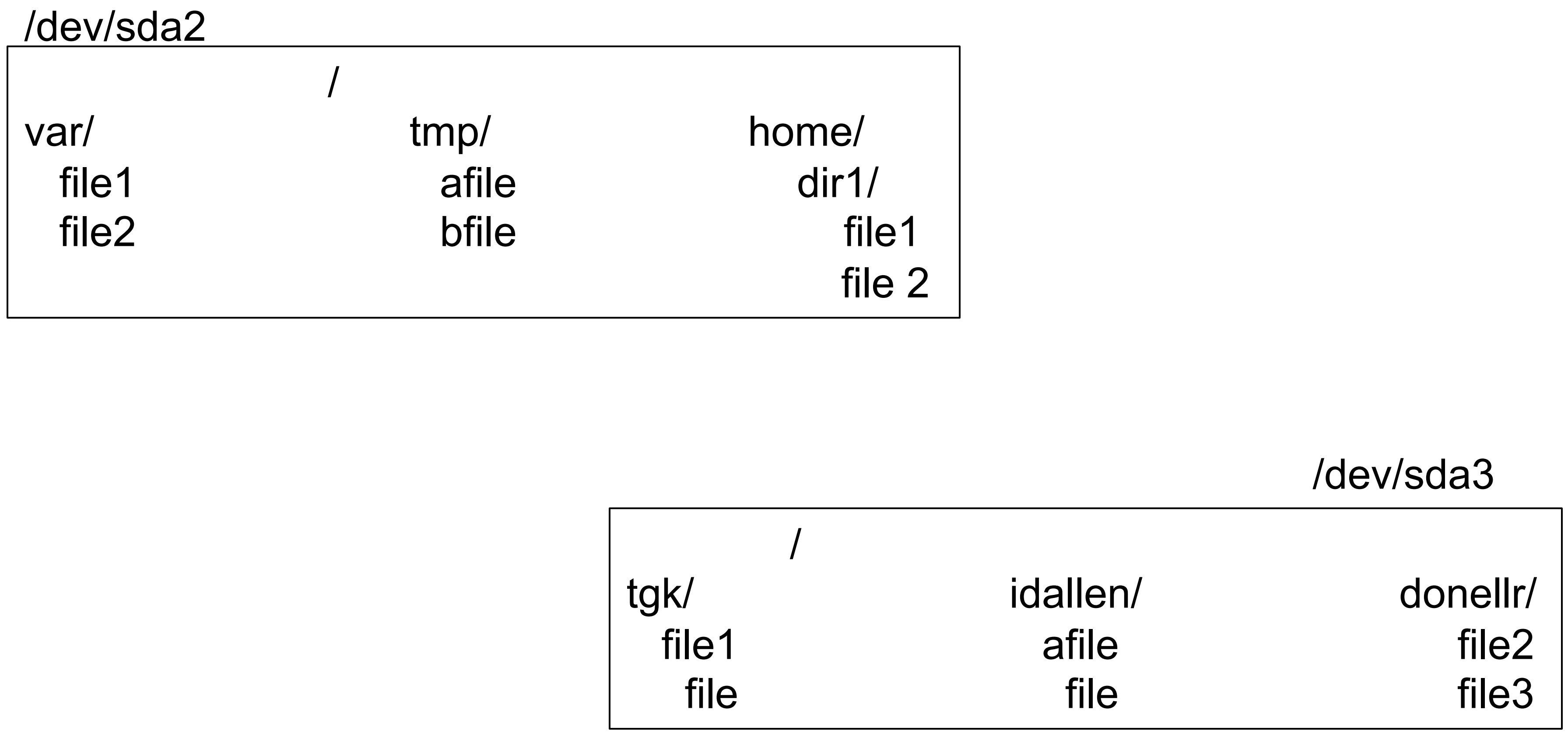
Linux/Unix mounting

- touch /home/donellr/file3



Linux/Unix mounting

- umount /dev/sda3



/etc/fstab

- man 5 fstab
- note that records for swap space appear in /etc/fstab, although swap space is not a filesystem (files are not stored in swap space)
- first field: device name
- second field: mount point
- third field: type
- fourth field: mount options
- fifth field: backup related (dump program)
- sixth field: file system check order

/etc/fstab (cont'd)

- mount options
  - on CentOS 5.8, "defaults" means
    - rw: read and write
    - dev: interpret device nodes
    - suid: setuid and setgid bits take effect
    - exec: permit execution of binaries
    - auto: mount automatically due to "mount -a"
    - nouser: regular users cannot mount
    - async: file I/O done asynchronously
- other options:
  - these are for quota utilities to see rather than mount
    - usrquota
    - grpquota

dmesg: kernel ring buffer

- http://teaching.idallen.com/cst8207/13w/notes/580_system_log_files.html
- kernel messages are kept in a ring buffer
- common way to access the boot messages, including device discovery
- dmesg
- example: look for disk discovery:
  - dmesg | grep sd
- (another way): look at disks/partitions that the kernel knows about:
  - cat /proc/partitions

Adding a disk

- # migrating the /usr directory to be a separate partition on new disk
- shut down machine
- connect new disk to machine
- power on machine
- partition new disk (fdisk command)
- make filesystem in new partition (mkfs command)
- single user mode (shutdown command)
- ensure target directory is backed up
- move the target directory out of way (/usr to /usr1) (mv command)
- create the mount point (to replace dir we just moved, same name)
- mount new filesystem (mount command)
- /usr1/bin/rsync -aHv /usr1/. /usr   (notice where rsync is!)
- add a record for the new filesystem /etc/fstab
- exit, to return to runlevel 3
- remove /usr1 (content should be backed up)

lsof and fuser

- Note the difference between a mountpoint and a directory
  - mountpoint: both of these commands will apply to the entire filesystem mounted there
  - directory: both of these commands will apply to just that directory, not recursively every subdirectory underneath it

- summary of lsof:
  - http://www.thegeekstuff.com/2012/08/lsof-command-examples/

- fuser: similar in purpose to lsof
- examples:
  - fuser /mountpoint   # all processes using the filesystem mounted at /mountpoint
  - fuser /home/dir   # all processes using the directory dir
- summary of fuser:
  - http://www.thegeekstuff.com/2012/02/linux-fuser-command/

Bind mounts

- A bind mount is used to mount a directory onto a mount point: man mount
- use the “bind” option for the mount command

```
# mount –o bind /some/dir   /anotherdir
```

  - now /some/dir and /anotherdir are the same directory
- Be careful with bind mounts, because they make it possible to form cycles in the file system
- e.g. dangerous: "mount –o bind /home /home/user/dir"
  - serious repercussions for
    - rm –rf /home/user   # will remove all of /home
    - find /home/user   # will never stop
    - any program that recursively descends directories

Quotas

- https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_Linux/6/html/Storage_Administration_Guide/ch-disk-quotas.html
- Example: enabling quotas on /home
- /etc/fstab: usrquota,grpquota mount options for file system containing /home
- quotacheck -cug /home
  - c: don't read quota files, create new quota files
  - u: do user quotas
  - g: do group quotas
- edquota username or setquota -u user soft hard isoft ihard fs
- edquota -t   # edit grace period
- quotaon -vaug # turn quotas on
- repquota -a   # report on quotas
- quotaoff -vaug; quotacheck -vaug; quotaon -vaug #single user mode

Installation DVD for rescue mode / Live CD

- There are dangers associated with doing file system operations on "system directories" that might be used in system operation.
- For example, many programs will use the shared libraries in /usr/lib, which disappear if we move /usr

- Also, there may come a time when the system won't boot properly: MBR corrupted, bad entry in /etc/fstab, inconsistent / file system

Growing a filesystem

- That LVM tutorial link again:
  - http://www.howtoforge.com/linux_lvm
- Because Red Hat's installer used Disk Druid to set up LVM and installed the root file system on a Logical Volume, we can
  - add a hard disk
  - create a partition on that hard disk
  - # or, maybe we already had an unused partition, such as a reclaimed Windows partition
  - set up that partition as a physical volume
  - add that physical volume to our Volume Group
  - grow the Logical Volume on the Volume Group
  - grow the file system on that Logical Volume

Booting Sequence (CentOS)

- Power button pressed
- BIOS
- POST
- MBR : contains grub stage 1
- grub stage 1 : to find grub stage 2
- grub stage 2 : to launch kernel
- kernel running
- init process (PID 1) : consults inittab
- /etc/inittab
- /etc/init.d/rc.sysinit
- /etc/rc.d/rc 3   : assuming default runlevel 3

SysVinit

- /etc/init.d/*
  - these are scripts for starting, stopping, restarting services
- /etc/rc.d/rc.N.d/*   #where N is a runlevel
  - these are symbolic links to service's script
  - begins with K means service should not be running in that runlevel: call it with "stop" argument
  - begins with S means service should be running in that runlevel: call it with "start" argument
- chkconfig maintains these scripts

/etc/inittab

- /etc/inittab contains records of the form
  - id:runlevels:action:process
  - id: identifies an entry
  - runlevels: the runlevels in which the action should be taken
  - action: the action that should be taken
  - process: the process to be executed