Linux Shell Scripts - Fundamentals

Course Index

1. Introduction to Shell
2. Shell Basics
3. Shell Environment
4. Advanced Editing Tools (sed\awk)
5. Scripting
6. Useful Commands
7. Appendix – Advanced Commands

Course Objectives

- To become a **sophisticated Linux user**
- To be familiar with the **shell environment**
- To **develop shell scripts** (bash style)
- To understand useful **Linux commands**

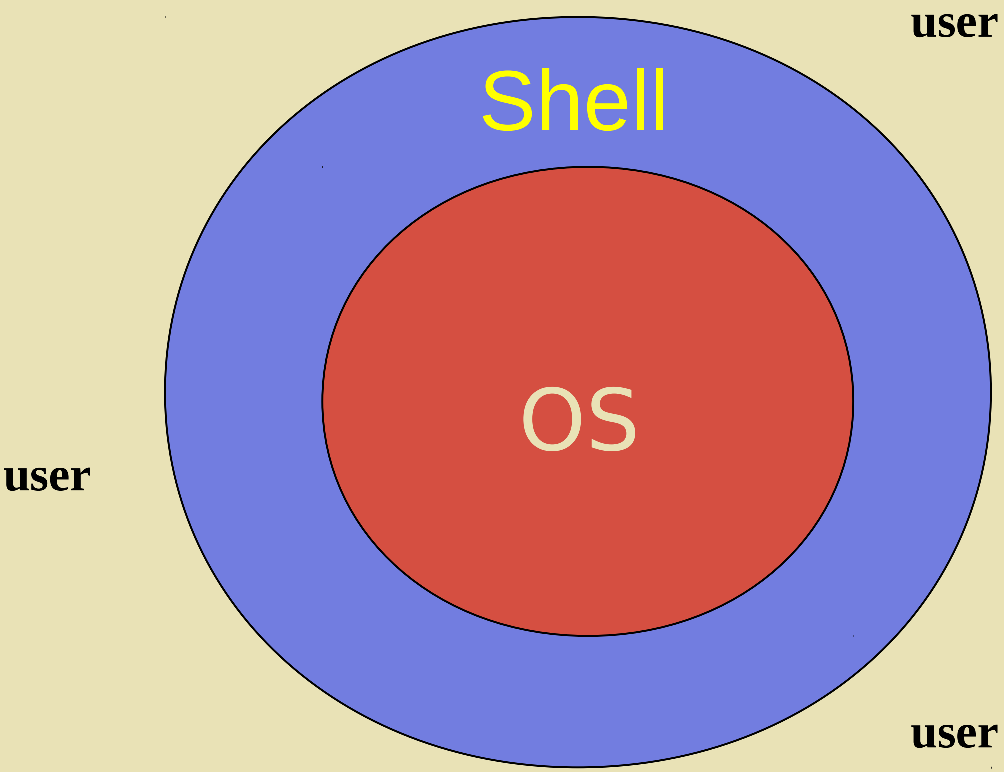
Chapter 1: Introduction to shell

LINUX
Your grandma can do it.

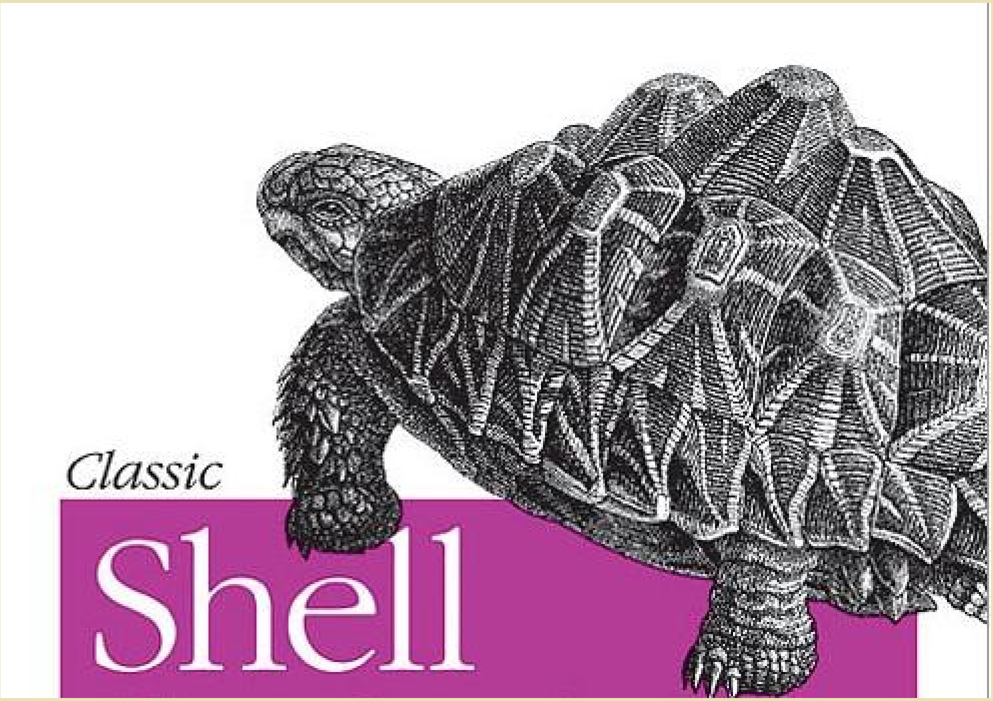
What is “the shell?”

- A program
- /bin/sh
- or maybe /bin/bash or /bin/csh or /bin/zsh or some other executable
- **Command line interface between the user and Linux Kernel**
- When the user logs in to the system the shell program is automatically executed

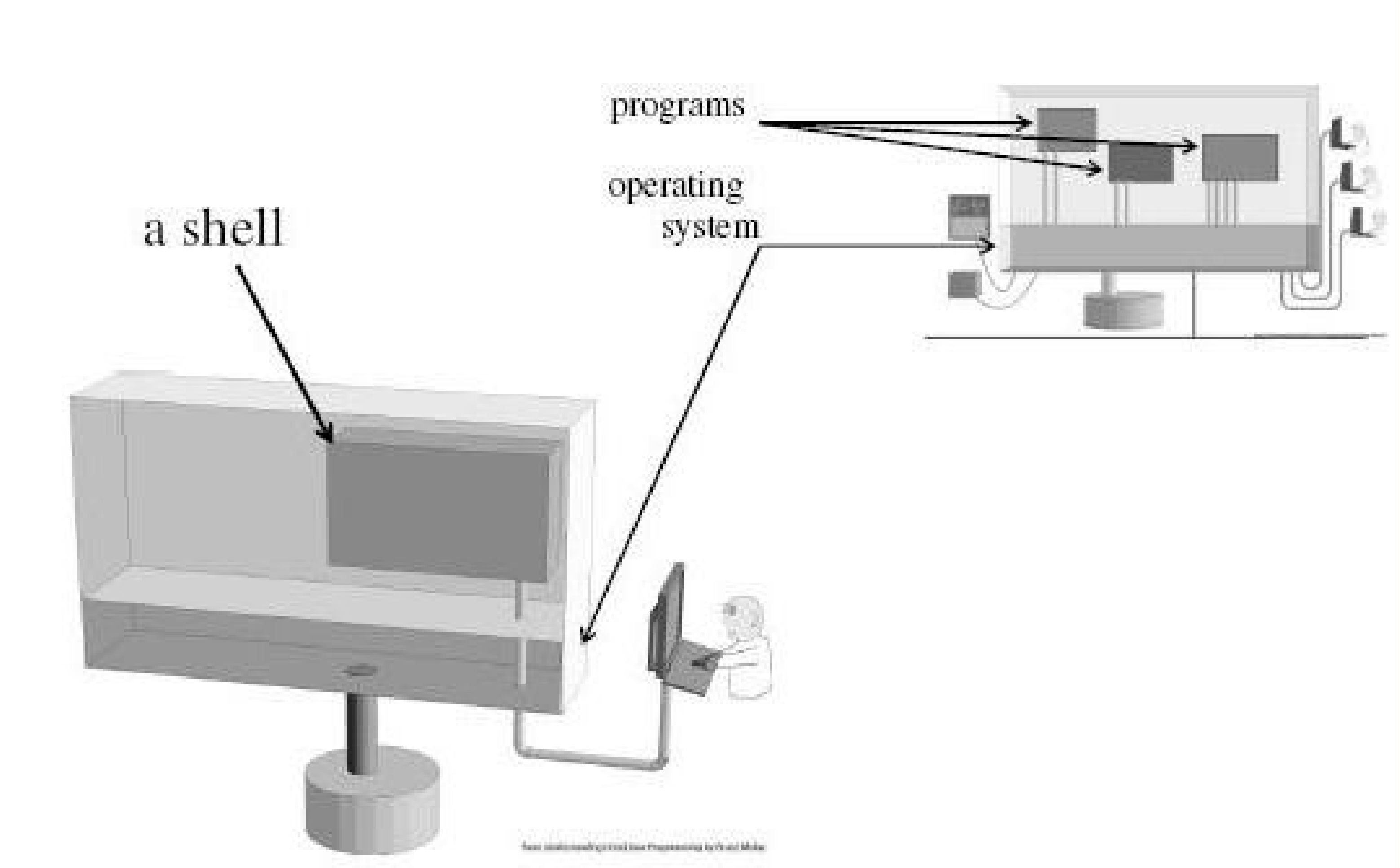
What is “the shell?”



Why is it called shell?

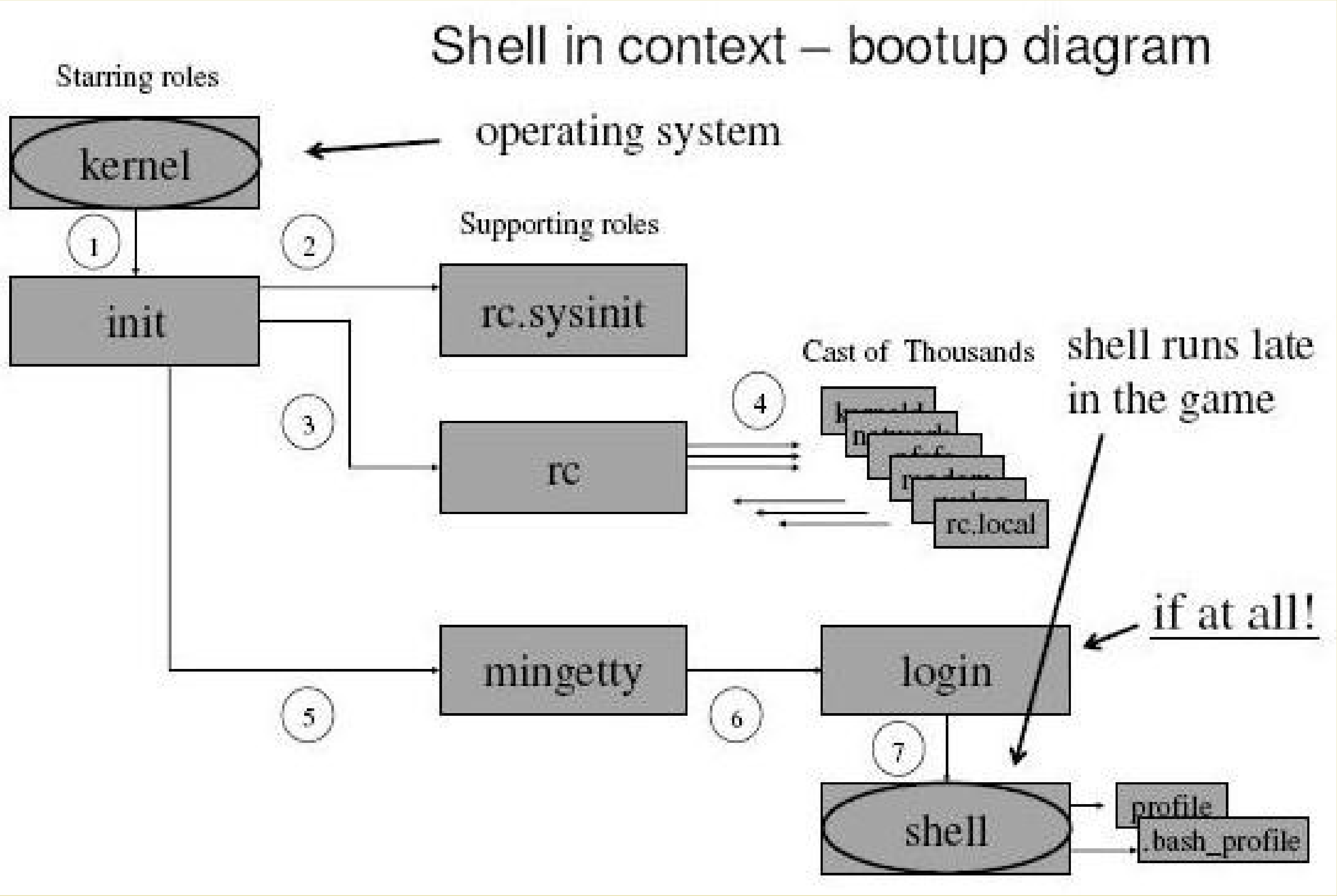


Where is the shell?



from Understanding Unix/Linux Programming by Bruce Molay

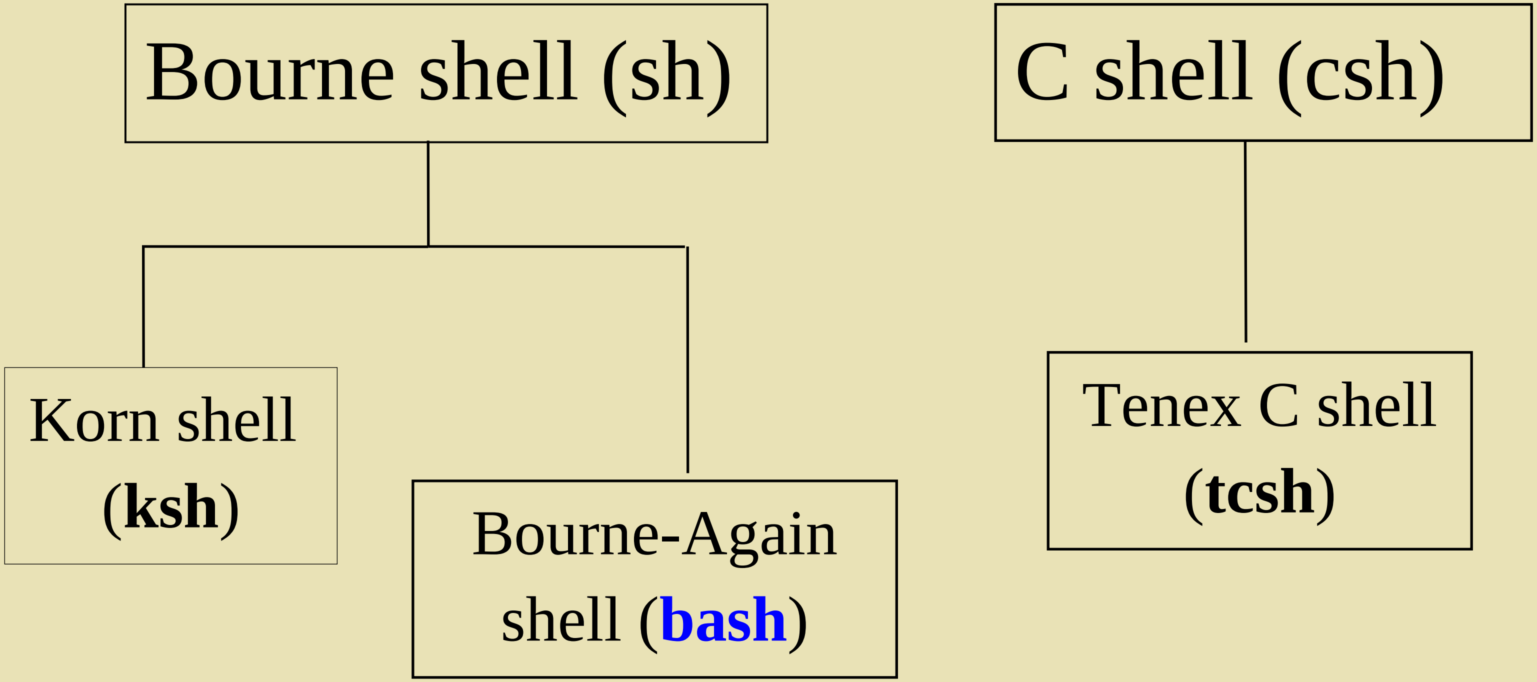
Where is the shell?



Shell – Features

- Command interpreter (parse and execute)
- I/O redirection
- Piping
- Environment control
- Background processing
- Shell scripts

Shell Flavors



Bourne shell (sh)

Korn shell (**ksh**)

Bourne-Again shell (**bash**)

C shell (csh)

Tenex C shell (**tcsh**)

What is your shell flavor?

- When the user logs in to Linux, he gets a **shell prompt, e.g #>**
- To identify the current shell flavor:
  #> echo $SHELL
- If you want to change your current shell to bash shell, type #> **bash**
- The **user default shell** flavor is defined inside the **/etc/passwd**

http://lirent.net

Chapter 2 : **Shell Basics**

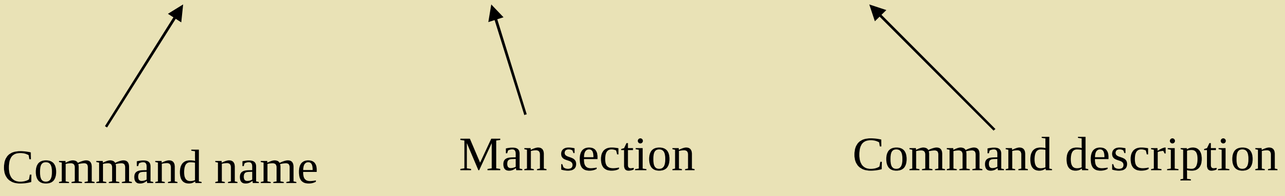
Chapter 2: Outline

- man - is the bible
- Wild Cards (ls, find…)
- Redirecting I/O
- Pipes and Filters
- Basic commands (head, tail, sort, e\grep)
- Commands listing
- Process Control (ps, kill, bg, fg, $$)
- Exercising

man - is the bible (1)

- # man ls
- # man -k "system disk space“

```
df                    (1)  - report file system disk space usage
```



Command name

Man section

Command description

# whatis du

```
du                    (1)  - estimate file space usage
```

man - is the bible (2)

- What is the difference?

```
# man 1 passwd
# man 5 passwd
```

- whatis passwd

```
passwd             (1)  - update a user's authentication tokens(s)
passwd             (5)  - password file
```

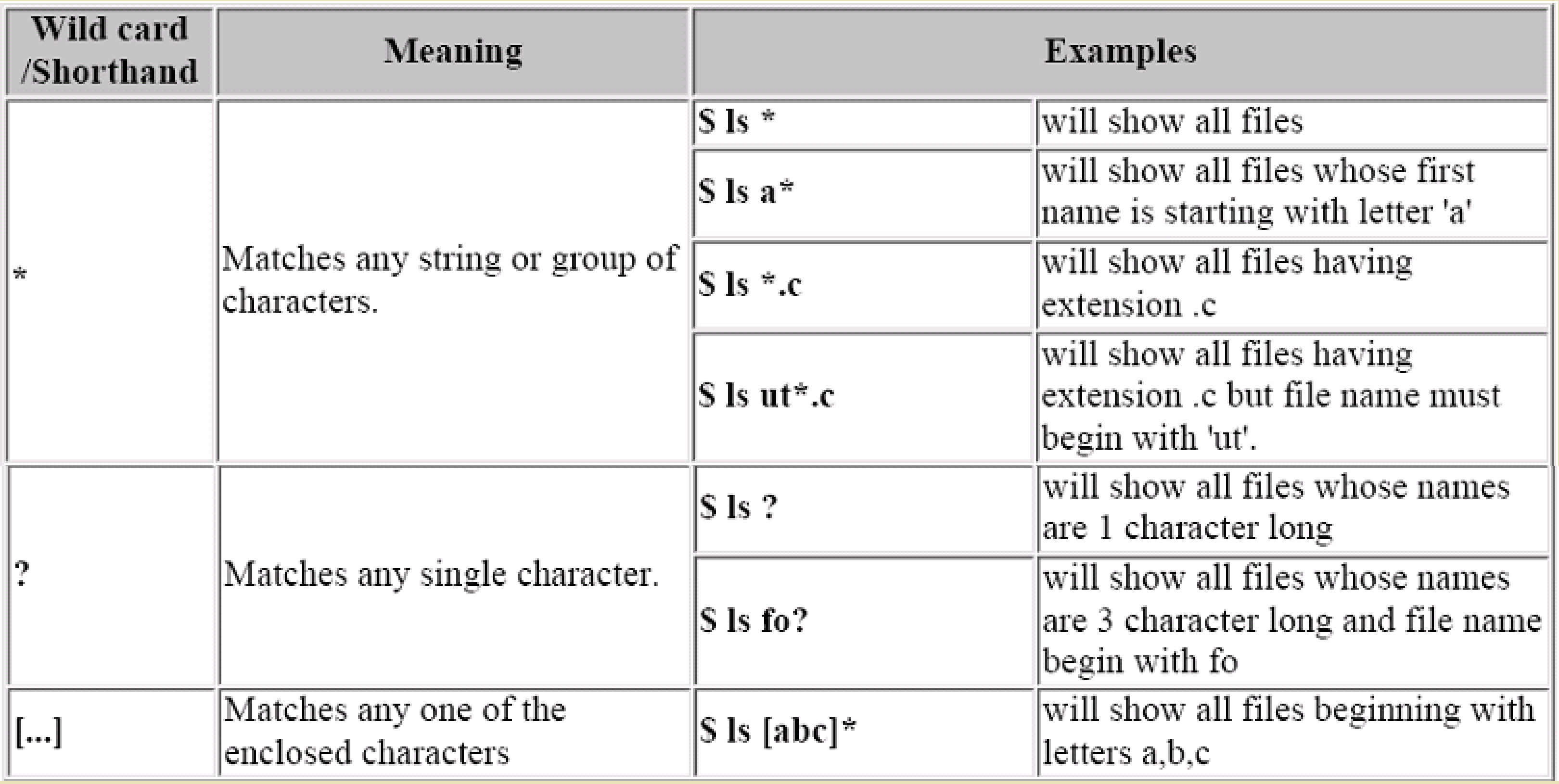
- How to build the whatis database?

```
# makewhatis
```

- RTFM ?

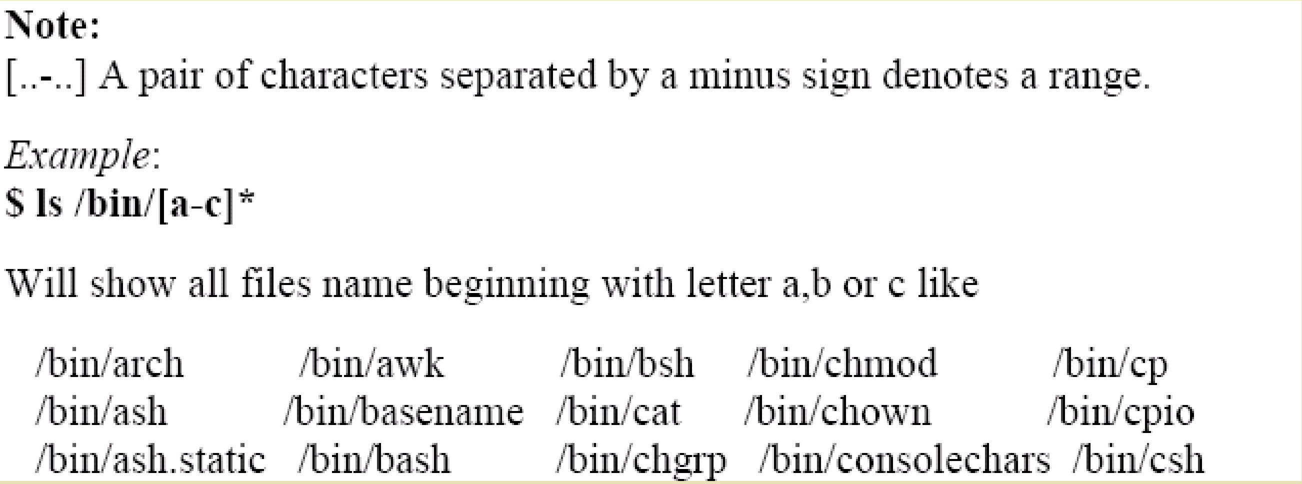
Wild Cards

- Make the command line much more powerful than any GUI file managers



Wild card /Shorthand	Meaning	Examples	
*	Matches any string or group of characters.	$ ls *	will show all files
		$ ls a*	will show all files whose first name is starting with letter 'a'
		$ ls *.c	will show all files having extension .c
		$ ls ut*.c	will show all files having extension .c but file name must begin with 'ut'.
?	Matches any single character.	$ ls ?	will show all files whose names are 1 character long
		$ ls fo?	will show all files whose names are 3 character long and file name begin with fo
[...]	Matches any one of the enclosed characters	$ ls [abc]*	will show all files beginning with letters a,b,c

Wild Cards - Examples



Note:

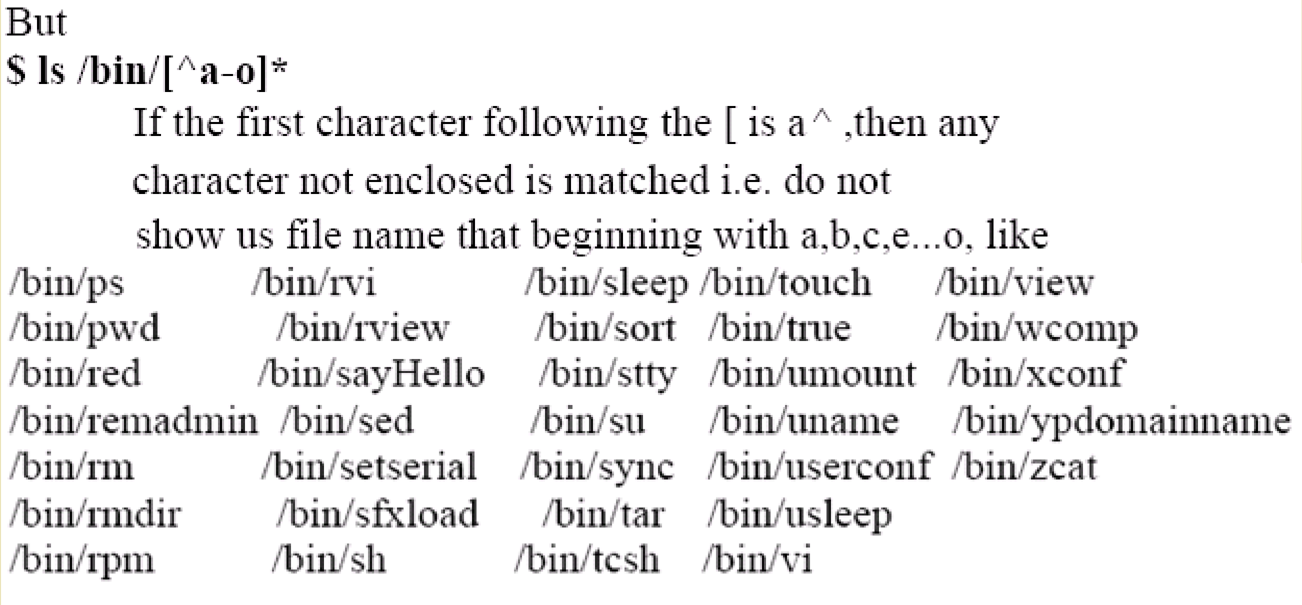
[..-..] A pair of characters separated by a minus sign denotes a range.

Example:

$ ls /bin/[a-c]*

Will show all files name beginning with letter a,b or c like

```
/bin/arch         /bin/awk        /bin/bsh     /bin/chmod          /bin/cp
/bin/ash         /bin/basename   /bin/cat     /bin/chown          /bin/cpio
/bin/ash.static   /bin/bash       /bin/chgrp   /bin/consolechars  /bin/csh
```



But

$ ls /bin/[^a-o]*

If the first character following the [is a ^ ,then any character not enclosed is matched i.e. do not show us file name that beginning with a,b,c,e...o, like

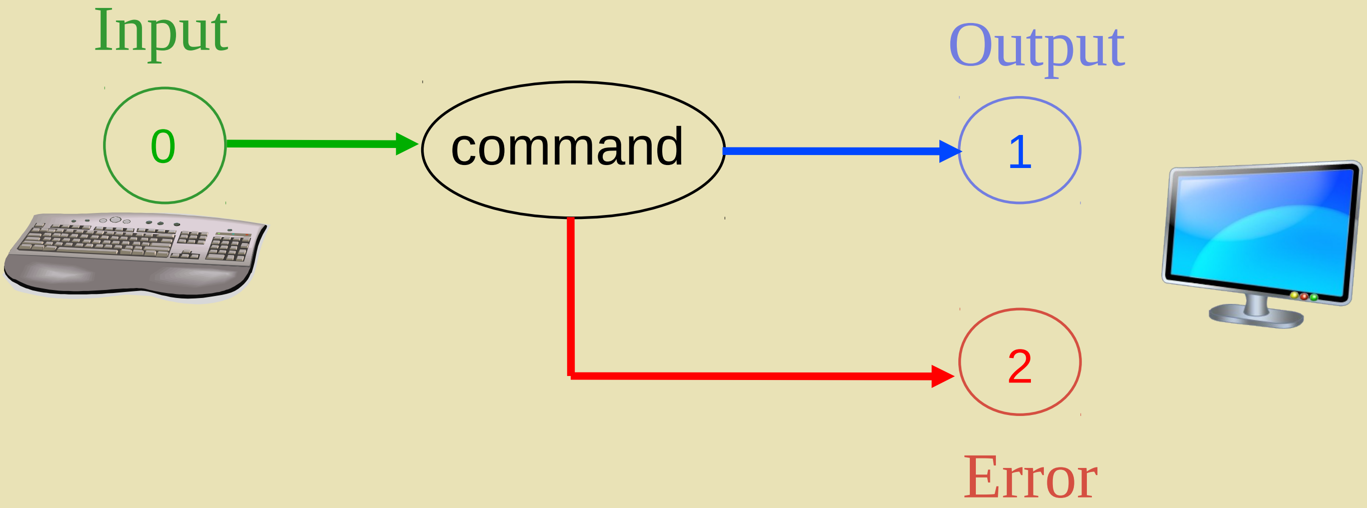
```
/bin/ps          /bin/rvi           /bin/sleep /bin/touch     /bin/view
/bin/pwd          /bin/rview        /bin/sort   /bin/true        /bin/wcomp
/bin/red          /bin/sayHello     /bin/stty   /bin/umount   /bin/xconf
/bin/remadmin  /bin/sed           /bin/su      /bin/uname    /bin/ypdomainname
/bin/rm           /bin/setserial   /bin/sync   /bin/userconf  /bin/zcat
/bin/rmdir        /bin/sfxload     /bin/tar     /bin/usleep
/bin/rpm          /bin/sh           /bin/tcsh    /bin/vi
```

Chapter 2: Outline

- man - is the bible
- Wild Cards (ls, find…)
- **Redirecting I/O**
- Pipes and Filters
- Basic commands (head, tail, sort, e\grep)
- Commands listing
- Process Control (ps, kill, bg, fg, $$)
- Exercising

Standard Input/Output/Error (1)

- Each process/command in the system has:
  - Standard input
  - Standard output
  - Standard error



Standard Input/Output/Error (2)

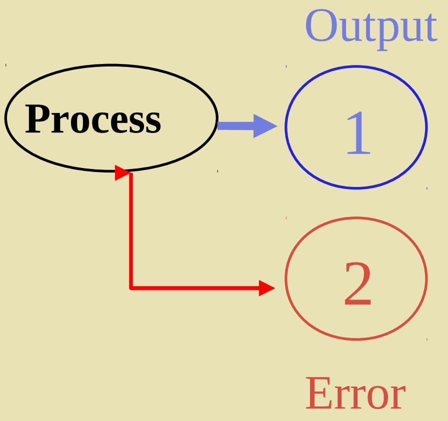
The file A exist

The file B is not exist

```
# ls  A  B
  ls: B: No such file or directory
  A
```

Stderr is terminal

Stdout is terminal

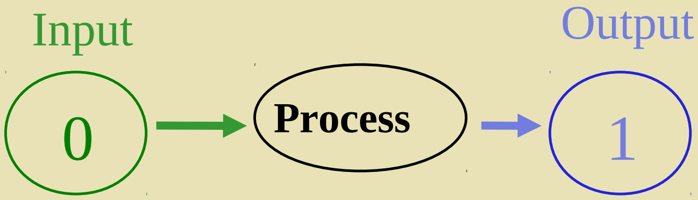


```
# tr [a-z] [A-Z]
  hello
  HELLO
```

Stdin is keyboard

Stdout is terminal

```
# cat
```



Redirection (1)

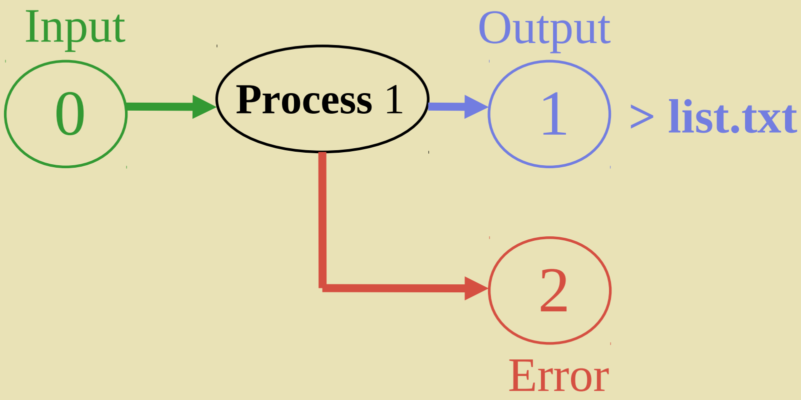
- What is redirection ?
  - Change the command *stdin*, *stdout or stderr*.
  - Can be changed to file or device (in UNIX devices are files anyway)

Redirection (2)

- Redirecting **output**
  - **ls** > **list.txt**

    Redirects output of **ls** to the file **list.txt** (overwrite)
  - **ls** >> **list.txt**

    Redirects output of **ls** to file **list.txt (**append)



Input

Output

0

Process 1

1

> **list.txt**

2

Error

  - **cat** > **out**

Redirection (3)

- Redirecting **input**
  - ***command [opt] < file1***
  - Input comes from file1 and not from **stdin**
  - E.g:
    - **cat file.txt** - file.txt is an argument to cat.
    - **cat < file.txt** - print the same <u>BUT</u> the cat redirects input to be from file.txt (the file.txt is not a parameter to the cat command)
    - Normally **cat** takes input from **stdin** and waits for user input

Redirection (4)

- I/O redirection can be combined

```
#> cat < file.txt > newfile.txt
```

  - Reads input from *file.txt* and writes the output to *newfile.txt*

- **stderr** redirection to a file

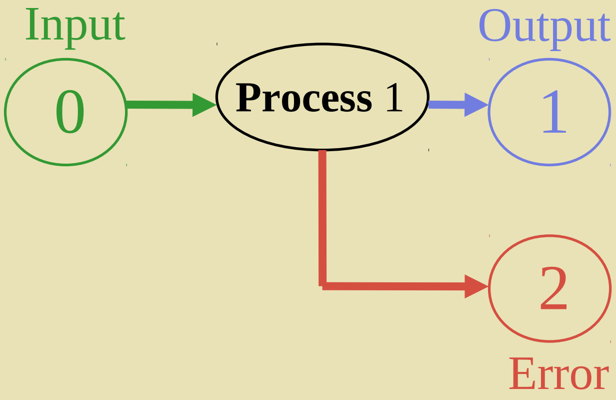
```
#> ls -l /tmp/not_exist 2> /tmp/ls_error
```

- **stderr** redirection to stdout which is a file

```
#> ls -l /tmp/not_exist > /tmp/out 2>&1
```

Redirection (5)

```
# touch A B
# rm B
```



```
# ls A B
  ls: B: No such file or directory
  A
```

Stderr is terminal

Stdout is terminal

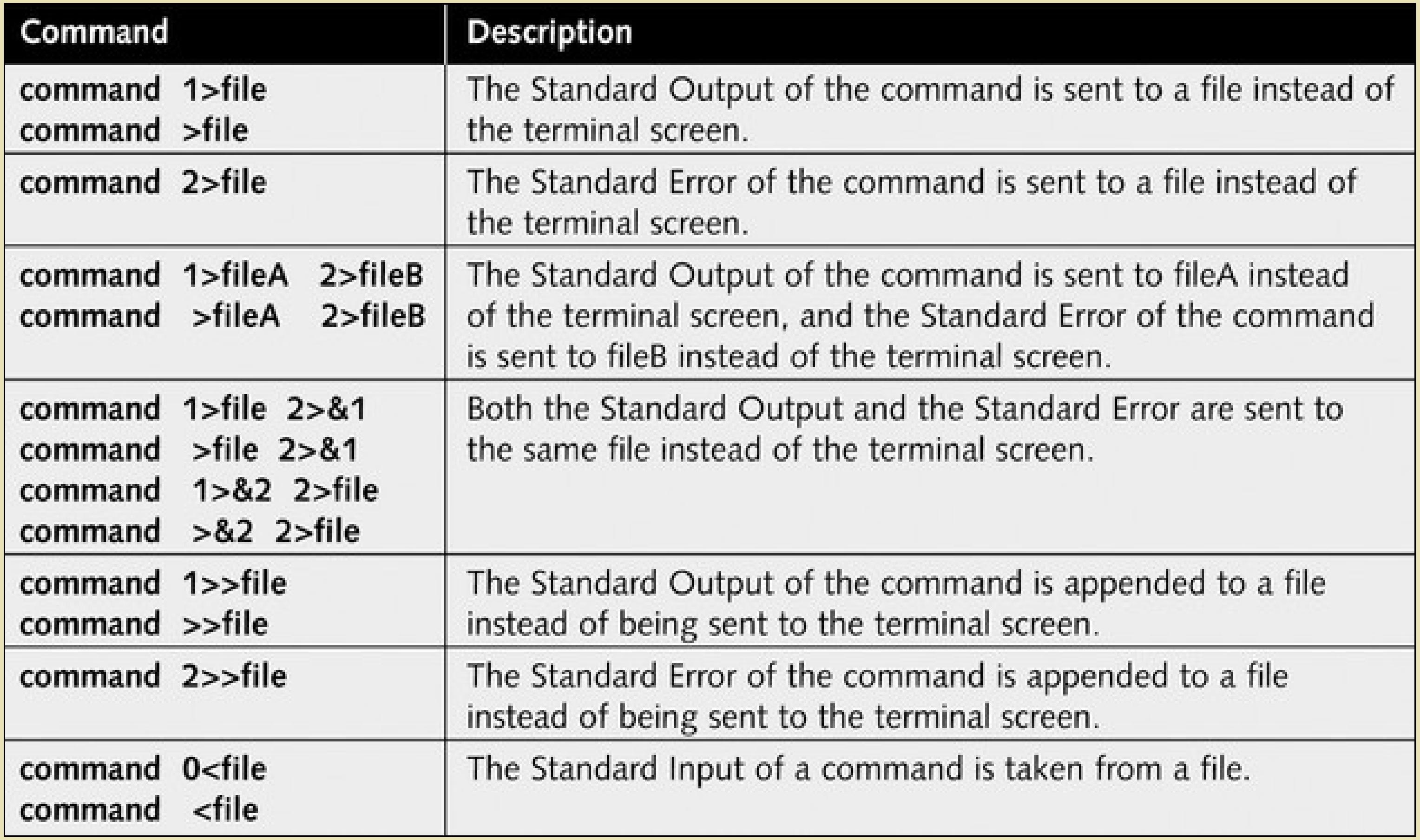
```
# ls A B > fileout
  ls: B: No such file or directory
```

```
# ls A B 2> filerr
  A
```

```
# ls A B > fileout_err 2>&1
```

Redirection (6)

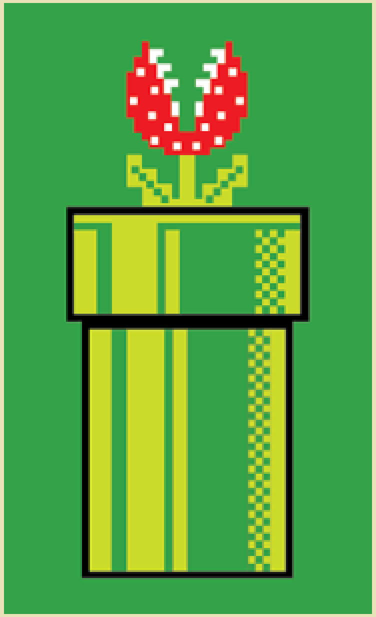
- Summary:



Command	Description
command 1>file command >file	The Standard Output of the command is sent to a file instead of the terminal screen.
command 2>file	The Standard Error of the command is sent to a file instead of the terminal screen.
command 1>fileA 2>fileB command >fileA 2>fileB	The Standard Output of the command is sent to fileA instead of the terminal screen, and the Standard Error of the command is sent to fileB instead of the terminal screen.
command 1>file 2>&1 command >file 2>&1 command 1>&2 2>file command >&2 2>file	Both the Standard Output and the Standard Error are sent to the same file instead of the terminal screen.
command 1>>file command >>file	The Standard Output of the command is appended to a file instead of being sent to the terminal screen.
command 2>>file	The Standard Error of the command is appended to a file instead of being sent to the terminal screen.
command 0<file command <file	The Standard Input of a command is taken from a file.

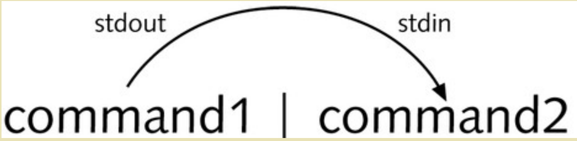
Pipes and Filters

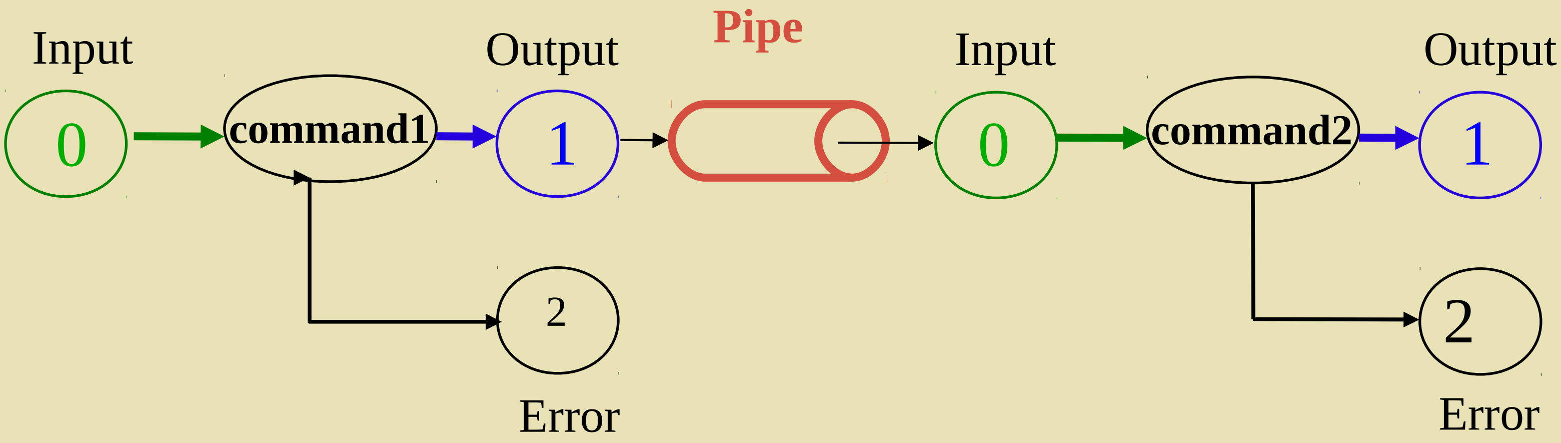
- What are Pipes ?



Pipes and Filters (1)

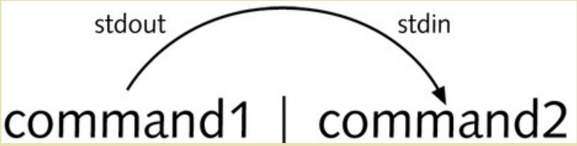
- What are Pipes ?
  - Pipes are mechanisms to connect the **stdout** of one command to the **stdin** of another.





Pipes and Filter (2)

- How to use a pipe ?



```
command1 | command2
```

#> ls | more

– Using redirection, we will have:

#> ls > tempfile

#> more tempfile

#> rm tempfile

Pipes and Filters (3)

- Another example:

```
#> ls –l | grep filename
```

  - **ls –l** gives a listing of the files, while piping it to **‘grep filename’** makes sure that we get the listing of only the files that we want.

```
#> find / | more
```

  - Show all the files in the systems screen by screen.

Pipes and Filters (4)

- Another example:

```
#> ls –l | tee /tmp/file_list
```

  - The **tee** command useful for printing the output of the last command and also save the output to a file

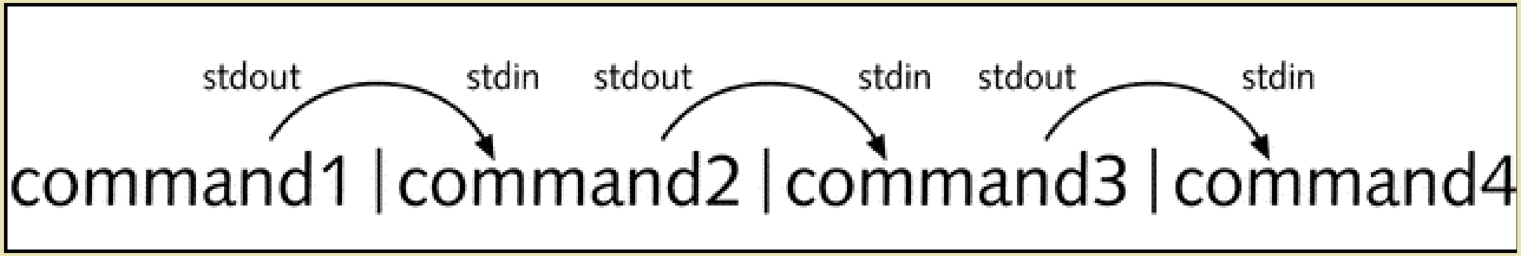
```
#> tar cvf  -  /oracle      |   gzip -c  > oracle.tar.gz
```

Without pipes:

```
# tar cvf oracle.tar /oracle
# gzip oracle.tar
```

Pipes and Filters (5)

- What is a filter ??
  - A command that processes and input stream to produce a modified output stream.
  - Filters are commands connected to other commands with pipes:



stdout stdin stdout stdin stdout stdin

command1 | command2 | command3 | command4

Pipes and Filters (6)

- E.g:
  - **cat** /var/log/messages **| grep** error **| grep** oracle
    - Show all the error lines in the log file.
  - **who | sort | more**
    - Produces a sorted listing of the users logged on to the system.
  - **ps –ef | grep** sleep | **grep –v** grep

Chapter 2: Outline

- man - is the bible
- Wild Cards (ls, find…)
- Redirecting I/O
- Pipes and Filters
- **Basic commands (head, tail, sort, e\grep)**
- Commands listing
- Process Control (ps, kill, bg, fg, $$)
- Exercising

head \ tail

- # **head -2** /proc/meminfo (first 2 lines)

```
MemTotal:      1545360 kB
MemFree:         27912 kB
```

- #cat /proc/meminfo | **head -2** :-)
- #cat < /proc/meminfo | **head -2** :-(

- # **tail -1** /proc/meminfo (last line)

```
DirectMap4M:     491520 kB
```

- # ls -lrt | **tail -3** (3 updated files)

- # **tail -f** /var/log/messages (watch the file)

Sort (1)

- Show processes sorted by name:

```
# ps -ef | sort -k 8 | tail -3
```

main column for sorting

```
xfs       1100    1   0 Sep13 ?        00:00:00 AmyProcess
root      1050   10  0 Sep13 ?        00:00:00 ABmyProcess
root       900    8    0 Sep13 ?        00:00:07 CmyProcess
```

- Show processes sorted by PID:

```
# ps -ef | sort -k 2                 :-(
# ps -ef | sort -k 2 -n                   :-)
```

numerical sort

```
root       900    8    0 Sep13 ?        00:00:07 CmyProcess
root      1050   10  0 Sep13 ?        00:00:00 ABmyProcess
xfs       1100    1   0 Sep13 ?        00:00:00 AmyProcess
```

Sort (2)

- Only unique lines

```
# cat > /tmp/myfile
b
b
a
a
b
c
Ctrl-d
```

```
# sort –u /tmp/myfile
a
b
c
```

grep – Pattern Matching (1)

```
# cat >myfile
    # Test file for grep
    root  is the best user
    many users in /etc/passwd
    #  The tree have many roots
    I am not a root user.
    Ctrl-d
```

```
# grep root myfile
   root is the best user
   #  The tree have many roots
   I am not a root user.
```

```
# grep -n root myfile
   2:root is the best user
   4:#  The tree have many roots
   5:I am not a root user.
```

```
# grep -w root myfile
   root is the best user
   I am not a root user.
```

```
# grep -v root myfile
   # Test file for grep
   many users in /etc/passwd
```

```
# grep ^I myfile
   I am not a root user.
```

```
# grep s$ myfile
   #  The tree have many roots
```

```
# grep -i test myfile
   # Test file for grep
```

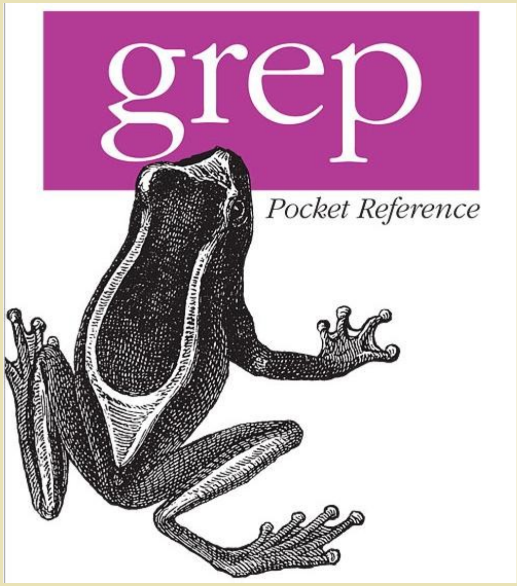
```
# egrep "no|test" myfile
   # Test file for grep
   I am not a root user.
```

grep – Pattern Matching (2)

- A useful command:

```
# cat /etc/hosts | grep -v "^#" | sort –u
```

- References (O’reilly books):



Cut

- Cut is a tool to cut sections from file.
- Examples

```
# echo “XIV System” | cut -d’ ‘  -f1
XIV

# echo “XIV System” | cut -d’ ‘ –f2
System


# echo “XIV:System” | cut -d: –f2
System

# grep bash /etc/passwd | cut -d: -f1
```

ssh (1)

- **ssh** is a command for logging or execute commands on remote machine.

  (It based on secure data communication protocol named SSH.)

- How to use it:

```
#> ssh   user@host
```

- **Login** into a remote machine

```
#> ssh   user@host   “command”
```

- **Execute commands** on a remote machine

ssh (2)

- How to do **Password-Less** SSH login

- In order to SSH password-less from *user*@host1 ==> *user*@host2, Login to host1 with *user* and run the following :

```
# ssh-keygen -t rsa          (use defaults & empty pass-phrase)

# ssh-copy-id  -i ~/.ssh/id_rsa.pub   user@host2
```

Note :

The first command create public\private keys.

The second command copy the public key to user@host2.

Now the SSH commands will not require password.

ssh (3)

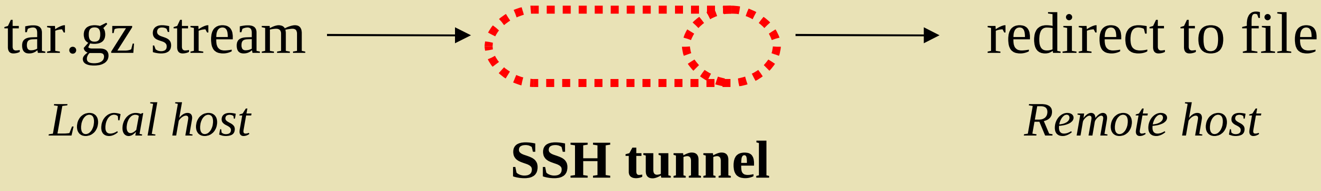
- How to backup directory?

Local backup

```
# tar cvf - /oracle | gzip -c | cat > oracle.tar.gz
```

Remote backup ?

```
# tar cvf - /oracle | gzip -c | ssh host1 “cat > oracle.tar.gz”
```



Commands Listing (1)

- cmd1; cmd2; ...

  Execute sequentially e.g:

  ```
  #> ./installOracle ; echo “installation finished ok?”
  ```

- cmd1 && cmd2

  Execute cmd2 if cmd1 has exit successfully e.g:

  ```
  #> ./installOracle && echo “installation finished ok!”
  ```

- cmd1 || cmd2

  Execute cmd2 only if cmd1 has non-zero exit status.

  ```
  #> ./installOracle || echo “failed to installation”
  ```

Commands Listing (2)

- cmd1 && cmd2 || cmd3

  Execute Combinations

  ```
  #> ./installOracle && echo “OK” || echo “FAILED”
  ```

Example :

```
#>ls file && echo “found” || echo “not found”
not found
 #>touch file
 #>ls file && echo “found” || echo “not found”
 found
```

Chapter 2: Outline

- Man is the bible
- Wild Cards (ls, find…)
- Redirecting I/O
- Pipes and Filters
- Basic commands (head, tail, sort, e\grep)
- Commands listing
- **Process Control(ps, kill, bg, fg, $$)**
- Exercising

Process Control (1)

- A program in execution is called a ***process***.
- Shell is a processes as well as Oracle and login sessions.
- Processes run in a hierarchical structure.
  - Root process (**called init)** -> parent **->** child.
  - Parent process **forks a child process.**
  - Since the **shell** is a process, it executes a command by forking a new process.

Process Control (2)

- Each process is assigned a unique Process ID (**PID**) by UNIX.
- Scheduling algorithms are used by the kernel to decide which process to run next.

Process Control (3)

- Processes can be run either in the **foreground** or the **background**.
- When a process is in the **foreground**, the shell is *waiting* for the process to finish.
- When a process is in the **background**, the shell is immediately available to the user.

- Adding the “**&**” to the end of the command executes it in the background.

Process Control (4)

```
# gunzip routing.ps.gz
#
# gunzip routing.ps.gz &
[1] 2703
#
```

Notes:

- [1] indicates the job number of the job and
- 2703 is the PID of the job.
- It’s better to redirect the output of backgrounf processes to a file or /dev/null.

Process Control (5)

- How do we know which processes are running ??
  - **ps [-aflu…] [PID] [UID]**
  - w/o PID display all user processes by default.
  - Options
    - **a**: show all processes in the system.
    - **f**: full list, medium info.
    - **l**: long list, detailed info.
    - **u**: list processes belonging to UID.

Process Control (6)

- E.g:

```
# ps

   PID TTY          TIME CMD
   775 pts/0    00:00:00 bash
  2407 pts/0    00:00:03 gunzip routing.ps.gz
  2758 pts/0    00:00:00 ps
```

<u>Important :</u>

#ps -H

#ps -fH

#ps -efH

Process Control (8)

- How do we show all background jobs ?
  - **jobs [-l…] jobid**
  - w/o jobid displays all background and suspended jobs.

Example :

```
# jobs
    [1]-  Running   gunzip public/html/routing.ps.gz &
```

Process Control (10)

- **Moving jobs** foreground and background.
  - If in foreground, suspend executing using **ctrl-z**.
  - Restart in background with **bg %jobnumber** (if % alone is used, then default is last job).
  - Restart in foreground with **fg %jobnumber**. (if % alone is used, then default is last job).

Process Control (11)

- How do we terminate a process ?
  - If job is in foreground, use **ctrl-c**
  - If job is in the background, bring to the foreground and then use **ctrl-c**
  - If **ctrl-c** does not work, then suspend the process with **ctrl-z**
    - Use **ps** to obtain the PID. Use the **kill** command to kill the process.

Process Control (11)

- For example :

```
 #> gzip /tmp/oracle
Ctrl-z
[1]+  Stopped                    gzip /tmp/oracle
#>
#> bg
[1]+ gzip /tmp/oracle &
#>
#> fg
gzip /tmp/oracle
Ctrl-c
#>
```

Process Control (12)

- Kill command
  - Send a signal to process.
  - **kill [-signal] PID**
  - Possible kill signals (> 30) include 3:quit, 9:absolute termination.
  - The most famous signal is -9, which kill the process for sure! #>**kill -9 <PID>**

Process Control (13)

- Signal list

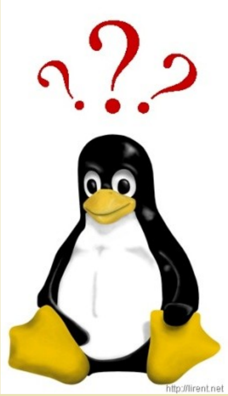
```
#> kill -l
 1) SIGHUP      2) SIGINT         3) SIGQUIT     4) SIGILL
 5) SIGTRAP     6) SIGABRT        7) SIGEMT      8) SIGFPE
 9) SIGKILL    10) SIGBUS        11) SIGSEGV    12) SIGSYS
13) SIGPIPE    14) SIGALRM       15) SIGTERM    16) SIGUSR1
17) SIGUSR2    18) SIGCHLD       19) SIGPWR     20) SIGWINCH
21) SIGURG     22) SIGIO         23) SIGSTOP    24) SIGTSTP
25) SIGCONT    26) SIGTTIN       27) SIGTTOU    28) SIGVTALRM
29) SIGPROF    30) SIGXCPU       31) SIGXFSZ    32) SIGWAITING
33) SIGLWP     34) SIGFREEZE     35) SIGTHAW    36) SIGCANCEL
```

Process Control (14)

- Kill example

```
#> csh
#> ps
   PID TTY      TIME CMD
  1580 pts/26   0:00 bash
  1730 pts/26   0:00 csh
#> kill -9 1730
Killed
#>
```

Chapter 2: Outline



- man - is the bible
- Wild Cards (ls, find…)
- Redirecting I/O
- Pipes and Filters
- Basic commands (head, tail, sort, e\grep)
- Commands listing
- Process Control (ps, kill, bg, fg)
- **Exercising (Chapter 1+2)**

Chapter 3: Shell Environments

Chapter 3: Outline

- Shell environment
- Variables (export, env)
- Startup .bashrc
- Source a file (.)
- Shell quotes (“ ’ `)
- Bash shortcuts
- Exercising

Shell Environment

- Shell environment
  - Consists of a set of **variables with values**.
  - These values are important information for the shell and the programs that run from the shell.
  - **You can define new variables** and change the values of the variables.

Shell Variables (1)

- **Built-in shell** variables:
  - **PATH**: The list of directories searched to find executables to execute.
  - **PS1** : The current **shell prompt**.
  - **SHELL**: The name of the login shell of the user.
  - **MANPATH**: Where man looks for man pages.
  - **LD_LIBRARY_PATH**: Where libraries for executables exist.

Shell Variables (2)

- **Built-in shell** variables…
  - **USER**: The user name of the user who is logged in to the system.
  - **HOME**: The user’s home directory.
  - **TERM**: The kind of terminal the user is using.
  - **DISPLAY**: Where X program windows are shown.

Shell Variables (3)

- How do we use the values in the shell variables?
  - Put a $ in front of their names to get the value inside.
  - For example

```
#>echo $SHELL
/bin/bash
#>echo ${SHELL}
/bin/bash
```

Shell Variables (4)

- Two kinds of shell variables:
  - **Local** variables
    - Not available in programs invoked from this shell.
  - **Environment** variables
    - Available in the current shell and the programs invoked from the shell

Shell Variables (5)

- Declaring **local** variables in *bash*:
  - **varname=varvalue**
  - No space between *varname* and *varvalue*.
  - Sets the variable *varname* to have value *varvalue*.

Shell Variables (6)

- Example:

```
# test="this is a test"
# echo $test
this is a test
# echo test
test
#
```

Shell Variables (7)

- Example with space b/w *varname* and *varvalue*. :-(

```
# val = "this is a test"
bash: val: command not found
# val= "this is a test"
bash: this is a test: command not found
#
```

Shell Variables (8)

- Remove declaration of **local** variables:
  - Use the unset command
    - Works for both the *C* shell and *bash* :-)
    - **unset *varname***
    - Once a variable is unset, the value that previously was assigned to that variable does not exist anymore

Example

```
# var="this is a test"
# echo $var
this is a test
# unset var
# echo $var

#
```

NOTE: Once the variable *var* is *unset*, there is no value that is part of the variable.

Shell Variables (9)

- Declaring **environment** variables in *bash*:
  - Using the **export** command.
  - To change a local variable to an environment variable, we need to *export* them.
  - **varname=varvalue**
  - **export varname**
  - Sets the environment variable *varname* to have value *varvalue*.

Example

```
# test="this is a test"
# export test
OR
# export test=“this is a test”
```

NOTE: The declaration with the *export* command can be combined into one statement as shown.

Shell Variables (10)

- Remove declaration of environment variables in *bash*:
  - Use the unset command.
  - **unset *varname***
  - Once variable is unset, the value that previously was assigned to that variable does not exist anymore.

Shell Variables (11)

```
# var="this is a test"
# export var
# echo $var
this is a test
#
# unset var
# echo $var

#
```

Shell Variables (12)

- We can use local variables, just like environment variables, so **why we have environment variables?**
  - Local variables are only available to the current shell.
  - Environment variables are accessible across shells and to all running programs.
    - What does this mean ? … examples follow.

Example (1)

```
# var="testing the variables"
# echo $var
testing the variables
#
# bash
# echo $var

#
```

NOTE: with the command *bash*, I invoke a new shell (*bash*) and in this shell, *var* is not accessible anymore.

Example (2)

```
# var="testing the variables“
# export var
# echo $var
testing the variables
#
# bash
# echo $var
testing the variables
#
```

NOTE: the environment variable is accessible even when I invoke another shell using the command *bash*.

Shell Variables (13)

- How to see all the **environment** variables in your shell:

```
# env
```

Shell Startup (1)

- Where to **define** the variables **permanently** (for every login)?

  - Many are defined in *.cshrc* and *.login* for the *C* shell and in ***.bashrc*** **and** ***.bash_profile*** for *bash*.

  - Inside shell scripts (later)

Shell Startup (2)

- When *csh* and *tcsh* are executed, they read and run certain configuration files:
  - **~/.login**:run once when you log in
    - Contains one time initialization, like TERM, HOME etc.
  - **~/.cshrc**: run each time another *csh/tcsh* process is invoked.
    - Sets lots of variables, like PATH, HISTORY, etc.
    - Aliases are normally written in this file.

Shell Startup (3)

- When *bash* is executed, it reads and runs certain configuration files:
  - **~/.bash_profile**(~/.profile): runs when you log in.
    - Contains one time initialization, like TERM, HOME etc.
  - **~/.bashrc**: run each time another *bash* process is invoked.
    - Sets lots of variables, like PATH, HISTORY, etc.

Shell Startup (4)

- Example ~/*.bashrc* file:

```
export PATH=${PATH}:/usr/oracle/scripts


admin=“root”
BOLD='\e[1m'
UNBOLD='\e[m'
PS1=“[\u@\h:$BOLD\w$UNBOLD]# “
```

e.g : [xavi@Barcelona:**/tmp**]#

Shell Startup (5)

- These files can be used for writing very useful commands.
  - Setting environment variables
  - Setting aliases
  - And more

Chapter 2: Outline

- Shell environment
- Variables (export, env)
- Startup .bashrc
- **Source a file (.)**
- Shell quotes (“ ’ `)
- Bash shortcuts
- Exercising

Source a File (1)

- It is also possible to define **variables inside a file** and source it in your shell (or in the .bashrc).
- Sourcing the file by running the command **dot**.
- Example :

```
#> cat > my_envs
myname=“XAVI”
PS1=‘XXX> '
Cntl-D
#> echo $myname

#
#> .  my_envs
XXX>
XXX> echo $myname
XAVI
```

Source a File (2)

- In a sourcing file you can also define aliases and functions (later).

Shell Quotes (1)

- Quotes in UNIX have a special meaning
  - **Double quotes**: “$myname”

    Shell variable expansion.

  - **Single quotes**: ‘$myname’

    Stops shell variable expansion.

  - **Back quotes**: `command`

    Replace the quotes with the result of the execution of the command.

Shell Quotes (2)

- Double and Single quotes:

```
# echo "Welcome $USER"
Welcome axgopala

# echo 'Welcome $USER'
Welcome $USER
#
```

Shell Quotes (3)

- Back quotes:

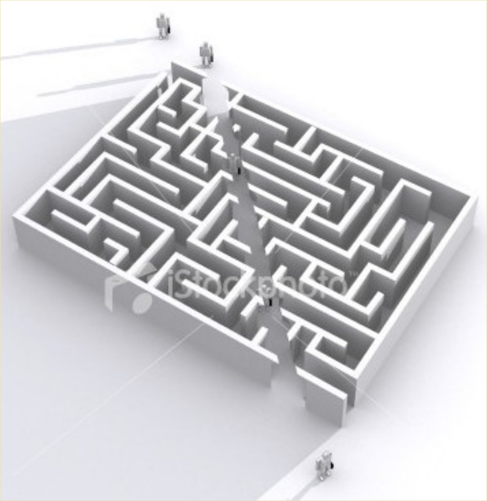
```
# var=`hostname`
# echo $var
nimni
#
```

```
# echo “HostName is `hostname`”
HostName is nimni
#
```

NOTE: The *hostname* command returns the name of the machine, which in this case is *nimni*

Bash Shortcuts

- Command\file completions:
  - First TAB -> Command completion.
  - Second TAB -> File name completion.
- Line movement
  - Cntl-a -> Move to the beginning of the line
  - Cntl-e -> Move to the end of the line
- History commands search
  - Cntl-r -> search commands in history
  - !! -> Run the last command
  - !<prefix> -> Run the last prefix command



Chapter 3: Outline

- Shell environment
- Variables (export, env)
- Startup .bashrc
- Source a file (.)
- Shell quotes (“ ’ `)
- Bash shortcuts
- **Exercising (Chapter 3)**

Chapter 4: Advanced Editing Tools (sed/awk)

Chapter 4: Outline

- Regular Expressions
- grep
- sed
- awk
- Exercising

Regular Expressions (regex)

- Pattern matching rules
- Used in grep, sed, awk , (everywhere?)
- Basic regex : ^$.*+?[]

^<string>	Match <string> in the beginning of the line.	
<string>$	Match <string> in the end of the line	
.	Matches any character, include newline	
<string>*	Matches a zero or more.	e.g. a* = , a, aa, aaa, etc
<string>\+	Matches a one or more.	e.g. a+ = a, aa, aaa, etc
\?	Matches only a zero or one	
[string]	Matches any single character in <string>	

- Note: Regex is much more powerful than Wild Cards.

grep - examples

file

```
The good
The bad
And The ugly
```

```
# cat file | grep “good”
   The good
```

```
# cat file | grep “^The”
   The good
   The bad
```

```
# cat file | grep “d$”
   The good
   The bad
```

```
# cat file | grep “o*”
   The good
   The bad
   And The ugly
```

```
# cat file | grep “o\+”
   The good
```

```
# cat file | grep “[au]”
   The bad
   And The ugly
```

sed (1)

- sed is a utility for **Editing file\streams** without using editor.
- For example, apply same changes to lots of source files.
- Not really a programming language.

sed (2)

- How to replace text in file?

```
#> sed [-i] ‘s/regex/replacement/g’ <file>
```

- For example:

```
#> cat teeORmilk
India's milk is good.
tea Red-Lable is good.(tee ?)
tea is better than the coffee.

#> sed ‘s/tea/milk/g' teaORmilk > /tmp/result.tmp.$$
#> cat /tmp/result.tmp.$$
India's milk is good.
milk Red-Lable is good.(milk ?)
milk is better than the coffee
```

sed - Examples

- Replace all digits: `sed ‘s/[0-9]/X/g‘ file` 13 => XX
- Replace all full num: `sed ‘s/[0-9]\+/X/g‘ file` 13 => X
- Replace in many files

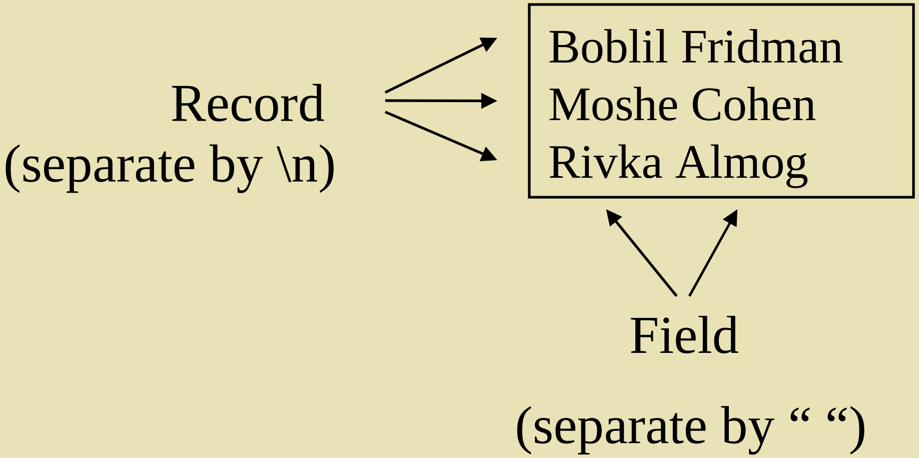
```
sed -i 's/X/Y/g'  *.sh
find . -name *.sh -type f -exec sed -i 's/X/Y/g' {} \;
find . -name *.sh -type f | xargs sed -i 's/X/Y/g'
```

- Delete lines 1-3: `cat /etc/hosts | sed ‘1,3d‘`
- Delete comments: `sed ‘/^#/d‘ /etc/hosts`

awk (1)

- awk is a **programming language** that is designed for processing text-based data, either in files or data streams.
- awk **reads from a file** or from its **standard input**, and outputs to its standard output.
- awk recognizes the concepts of **"file", "record"** and **"field".**

Input **file** e.g /tmp/worker



Boblil Fridman
Moshe Cohen
Rivka Almog

awk (2)

- /tmp/workers file for some awk example

```
Boblil Fridman
Moshe Cohen
Rivka Almog
```

- Prints the last name only

```
#> awk '{ print $2 }' /tmp/workers
Fridman
Cohen
Almog
```

```
#> cat /tmp/workers | awk '{ print $2 }'
```

awk – Examples (1)

- /etc/passwd file for some awk example

```
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
```

- Prints all the lines in /etc/passwd

```
#> awk '{ print $0 }' /etc/passwd
```

- Prints the 1st and 3rd fields of each line in /etc/passwd:

```
#> awk -F":" '{ print "username: " $1 "\t\t uid:" $3 }' /etc/passwd
```

awk – Examples (2)

- Print the size of all the files in /etc/* (only if it bigger then 5MB)

```
#>  du -ks /etc/* | awk 'BEGIN {count=0} \
                {count=count+$1} \
        END { size=count/1024; if (size>5) {print size "MB"}}'
```

```
#> du –ks /etc/* |head -3
20      /etc/a2ps.cfg
8      /etc/a2ps-site.cfg
40      /etc/acpi
```

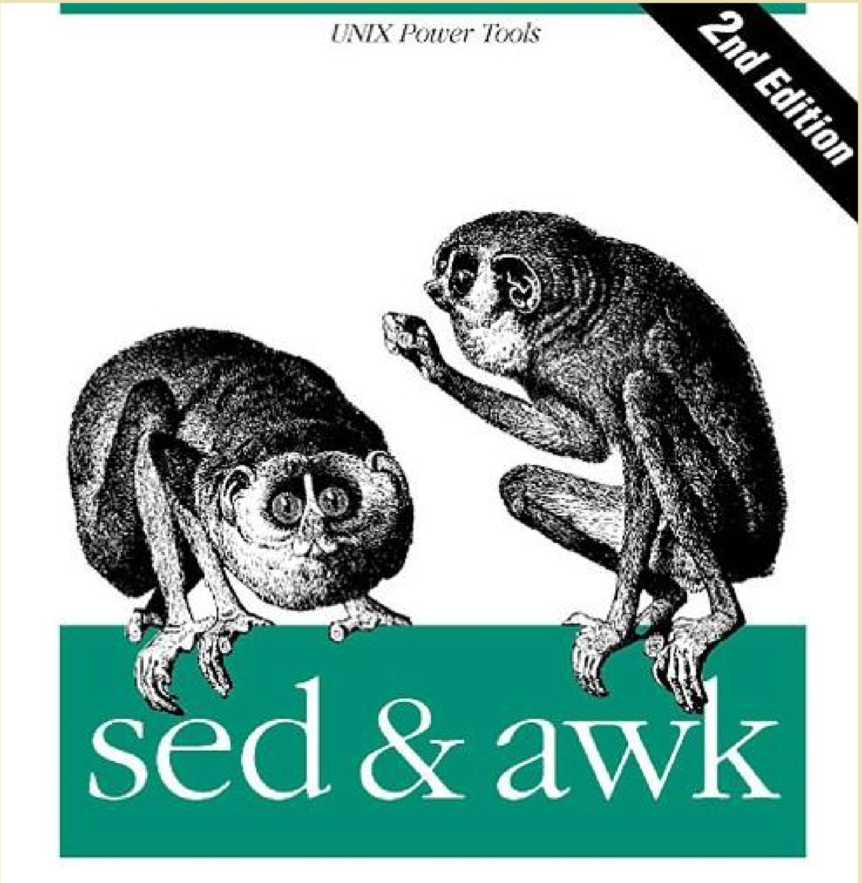
- Same same but with awk file

```
#>  du -ks /etc/* | awk –f /tmp/awk_file
```

```
#> cat /tmp/awk_file
BEGIN{count=0}
{count=count+$1}
END{
   size=count/1024;
   if (size>5) {print size "MB“}
}
```

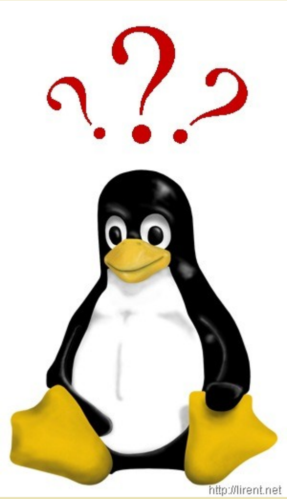
sed and awk

- References (O’reilly books):



Chapter 4: Outline

- Regular Expressions
- sed
- awk
- **Exercising**



Chapter 5: Scripting

Chapter 5: Outline

- What is a script?
- How to run scripts
- Variables
- Shell arithmetic's (expr)
- Controls (for, if, test, files\string\num)
- Exercising A
- Controls cont… (while, case)
- Arguments ($0, $*, $?, $$)
- Reading input (read)
- Functions
- Traps (trap “” SIGINT)
- Debug a script (bash -x \ set -x)
- Exercising B

Shell Scripts (1)

- A shell script is a **text file with UNIX commands** in it.

- The first script is /tmp/**hello.sh** which contain two lines:

```
#!/bin/bash
echo “Hello world”
```

Shell Scripts (2)

- Shell scripts usually begin with a **#!** and a shell name.
  - Pathname of shell be found using the ***which*** command.
  - The shell name is the shell that will execute this script.
    - For example, #!/bin/bash
- **If no shell is specified** in the script file, the current shell will be chosen.

Shell Scripts (3)

- Any UNIX command can go in a shell script
  - Commands are **executed in order or** in the flow determined by control statements.
- Different shells have different control structures
  - We will **focus on bash**.

Shell Scripts (4)

How to run shell scripts

- Shell script should be with executable **permissions**
  - Must use *chmod* to change the permissions of the script to be executable.
  - For example, #> **chmod u+x /tmp/hello.sh**
- **Execute the script** by absolute path:

```
#> /tmp/hello.sh
Hello world
```

- Or by relative path:

```
#> cd /tmp
#> ./hello.sh
Hello world
```

Shell Scripts (5)

- It is also possible to execute the script by specifying the shell name.
  - For example, **$ bash** hello.sh
  - For example, **$ csh** hello.csh

Note : We use this way mostly during debug of a script (later)

Shell Scripts (6)

- Why write shell scripts?
  - **To avoid repetition**:
    - If you do a sequence of steps with standard UNIX commands over and over, why not do it all with just one command?
    - Or in other words, store all these commands in a file and execute them one by one.

Shell Scripts (7)

- Why write shell scripts?
  - **To automate difficult tasks**:
    - Many commands have subtle and difficult options that you do not want to figure out or remember every time.

For example:

http://www.youtube.com/watch?v=bYcF_xX2DE8&feature=fvw

Simple Example (1)

- Assume that I need to execute the following commands once in a while when I run out of disk space:

```
#> rm -rf $HOME/.netscape/cache
#> rm -f $HOME/.netscape/his*
#> rm -f $HOME/.netscape/cookies
#> rm -f $HOME/.netscape/lock
#> rm -f $HOME/.netscape/.nfs*
#> rm -f $HOME/.pine-debug*
#> rm -fr $HOME/nsmail
```

Simple Example (2)

- We can put all those commands into a shell script, called *myscript*.

```
#! /bin/bash
rm -rf $HOME/.netscape/cache
rm -f $HOME/.netscape/his*
rm -f $HOME/.netscape/cookies
rm -f $HOME/.netscape/lock
rm -f $HOME/.netscape/.nfs*
rm -f $HOME/.pine-debug*
rm -fr $HOME/nsmail
```

Sample Example (3)

- To run the script:
  - Step 1:
    - #> chmod u+x myscript

      (Only once)
  - Step 2:
    - Run the script:
    - #> ./myscript
- Each line of the script is processed in order.

Shell Scripts (8)

- Common editors for writing shell scripts:

– **vi**	**-**	**Text editor**
– **vim**	**-**	**Enhanced text editor**
– **gedit**	**-**	**GUI editor**
– **nedit**	**-**	**GUI editor**
– **editPlus**	**-**	**Windows GUI (edit remote files)**

Shell Scripts (9)

- Shell variables:
  - Declared by:

```
varname=varvalue
```

  - To make them an environment variable, we ***export*** it.

```
export varname=varvalue
```

Shell Scripts (10)

- Assigning the output of a command to a variable:
  - Using **back-quotes**, we can assign the output of a command to a variable:

```
#! /bin/bash
filelist=`ls`
echo $filelist
```

Shell Scripts (11)

- Example:

```
#ls
a  b  c  html/
# filelist=`ls`
# echo $filelist
a b c html/
#
```

Shell Scripts (12)

- The **expr** command:
  - Calculates the value of an expression.
  - For example:

```
# value=`expr 1 + 2`
# echo $value
3
#
```

Notes on *expr* (1)

- Why do we need the **expr** command?
  - For example:

```
# file=1+2
# echo $file
1+2
#
```

NOTE: 1+2 is copied as it is into *val* and not the result of the expression, to get the result, we need **expr**.

Notes on *expr* (2)

- Variables as arguments:

```
# count=5
# count=`expr $count + 1`
# echo $count
6
#
```

NOTE:

- *count* is replaced with its value by the shell!

- Another why : count=$(($count +1))

Notes on *expr* (3)

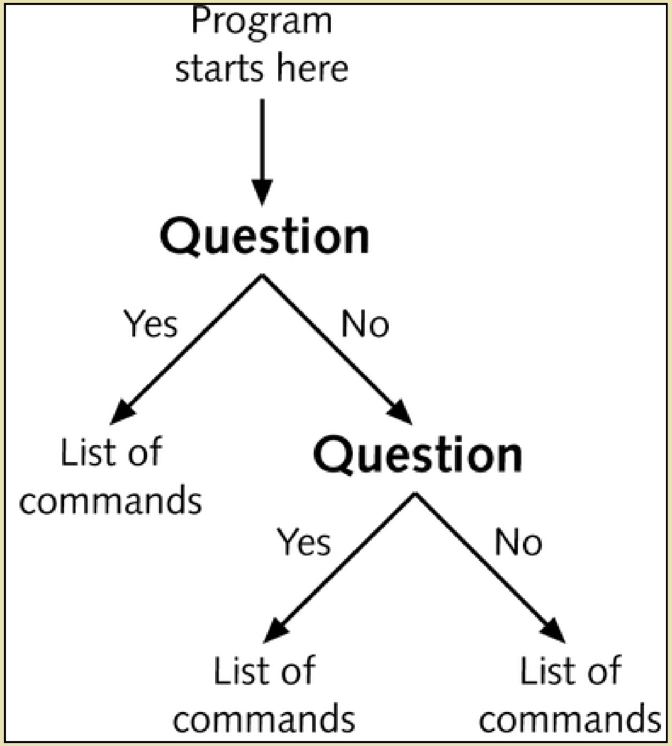
- expr supports the following operators:
  - arithmetic operators: +,-,*,/,%
  - comparison operators: <, <=, ==, !=, >=, >
  - boolean/logical operators: &, |
  - parentheses: (,)
  - precedence is the same as C, Java

Chapter 5: Outline

- What is a script?
- How to run scripts
- Variables
- Shell arithmetic's (expr)
- Controls (for, if, test, files\string\num)
- Exercising A
- Controls cont… (while, case)
- Arguments ($0, $*, $?, $$)
- Reading input (read)
- Functions
- Traps (trap “” SIGINT)
- Debug a script (bash -x \ set -x)
- Exercising B

Control Statements (1)

- Control statements control the flow of programs (script).



Control Statements (2)

- The most common types of control statements:
  - conditionals: **if / else / elsif, case, ...**
  - loop statements: **for, while**, until, do, ...

The *if* condition (1)

- Simple form:

```
if [ test ]; then
    command1
    command2
    …
fi
…
```

<u>There are 4 test types:</u>

1. Numeric tests: e.g [$num -eq 4]
2. string tests   : e.g [“$name” = “yosi”]
3. file tests   : e.g [-f /tmp/log]
4. variable tests : e.g [-n $var]

The *if* condition (2)

- The simplest flow control statement is the if condition.

```
#> age=29
#> if [ $age –lt 30 ]; then
>echo “You are under 30 years.”
> fi
  You are under 30 years.
```

Numeric test

If, elseif and else (1)

```
#!/bin/bash
age=60
if [ $age –lt 30 ]; then
    echo “You are under 30”
elif [ $age –gt 30 -a $age –le 40 ]; then
    echo “You are in your 30s”
else
    echo “You are 40 or over”
fi
```

is like AND operation.

test AND test

If, elseif and else (2)

- Few syntax to the same script.

```
#!/bin/bash

myvar="myvalue"
if [ "$myvar" = "" ]; then
     echo "nothing!";
else
     echo "got $myvar"
fi
```

```
#!/bin/bash

myvar="myvalue"
if [ "$myvar" = "" ]
then
     echo "nothing!";
else
     echo "got $myvar"
fi
```

```
#!/bin/bash

myvar="myvalue"
[ "$myvar" = "" ] && echo "nothing“ || echo "got
$myvar"
```

If, elseif and else (3)

- Few syntax to the same script.

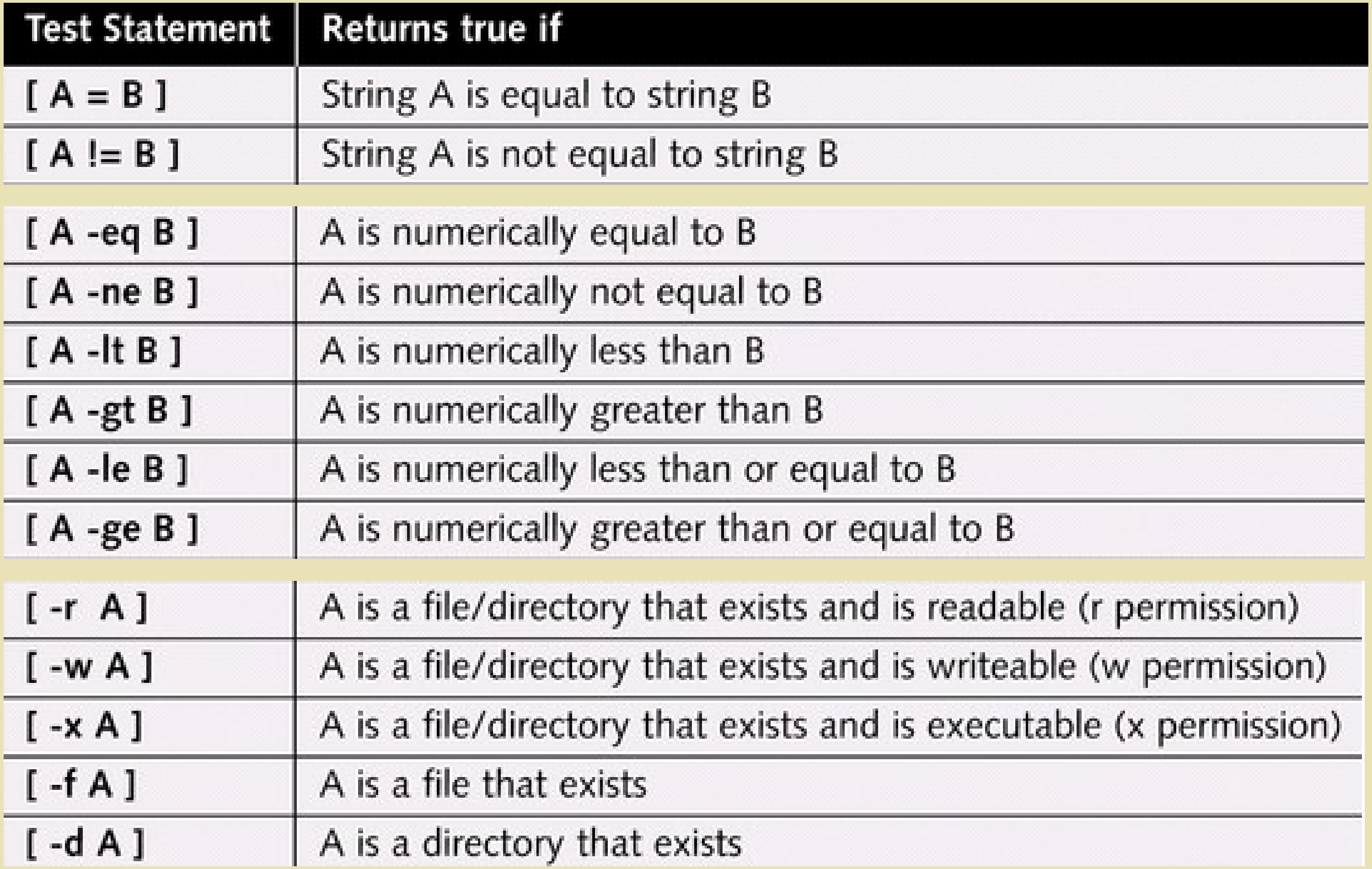
```
#!/bin/bash

myvar="myvalue"
if [ "$myvar" = "" ];
then
     echo "nothing!";
else
     echo "got $myvar"
fi
```

```
#!/bin/bash

myvar="myvalue"
if [ -z “$myvar” ]; then
     echo "nothing!";
else
     echo "got $myvar"
fi
```

Variable test

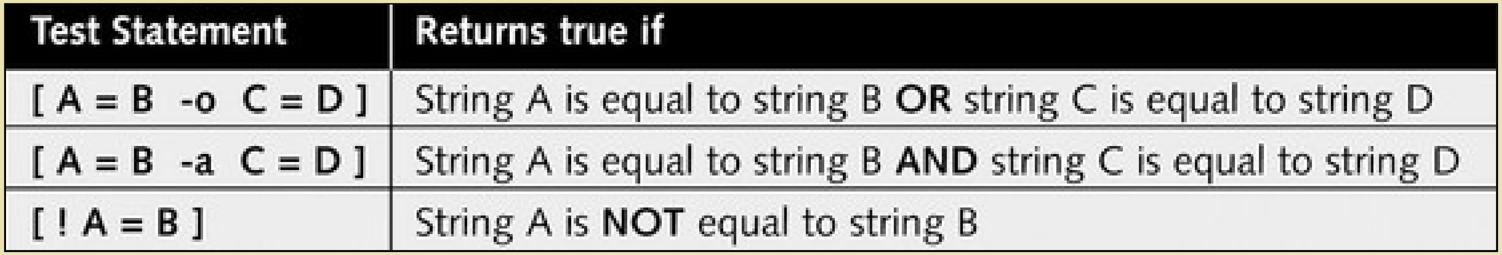
Summary of Test Types

string, numeric and files



Test Statement	Returns true if
[A = B]	String A is equal to string B
[A != B]	String A is not equal to string B
[A -eq B]	A is numerically equal to B
[A -ne B]	A is numerically not equal to B
[A -lt B]	A is numerically less than B
[A -gt B]	A is numerically greater than B
[A -le B]	A is numerically less than or equal to B
[A -ge B]	A is numerically greater than or equal to B
[-r A]	A is a file/directory that exists and is readable (r permission)
[-w A]	A is a file/directory that exists and is writeable (w permission)
[-x A]	A is a file/directory that exists and is executable (x permission)
[-f A]	A is a file that exists
[-d A]	A is a directory that exists

Test Special Operations



Test Statement	Returns true if
[A = B -o C = D]	String A is equal to string B **OR** string C is equal to string D
[A = B -a C = D]	String A is equal to string B **AND** string C is equal to string D
[! A = B]	String A is **NOT** equal to string B

- To see more tests example

  #> man bash

  And then search for : **/-a file**

Exit Code (1)

- Every well behaved command returns back a exit code.
  - **0 => successful**
  - 1..255 => unsuccessful
  - This is different from C \ Java.
  - The **exit code of the last command** is stored in the **$?** Environment variable.

Exit Code (2)

- For example:

```
#> ls /tmp/file_exist >/dev/null 2>&1
#> echo $?
0
#> ls /tmp/file_not_exist >/dev/null 2>&1
#> echo $?
2
#> echo $?
0
```

Exit Code (3)

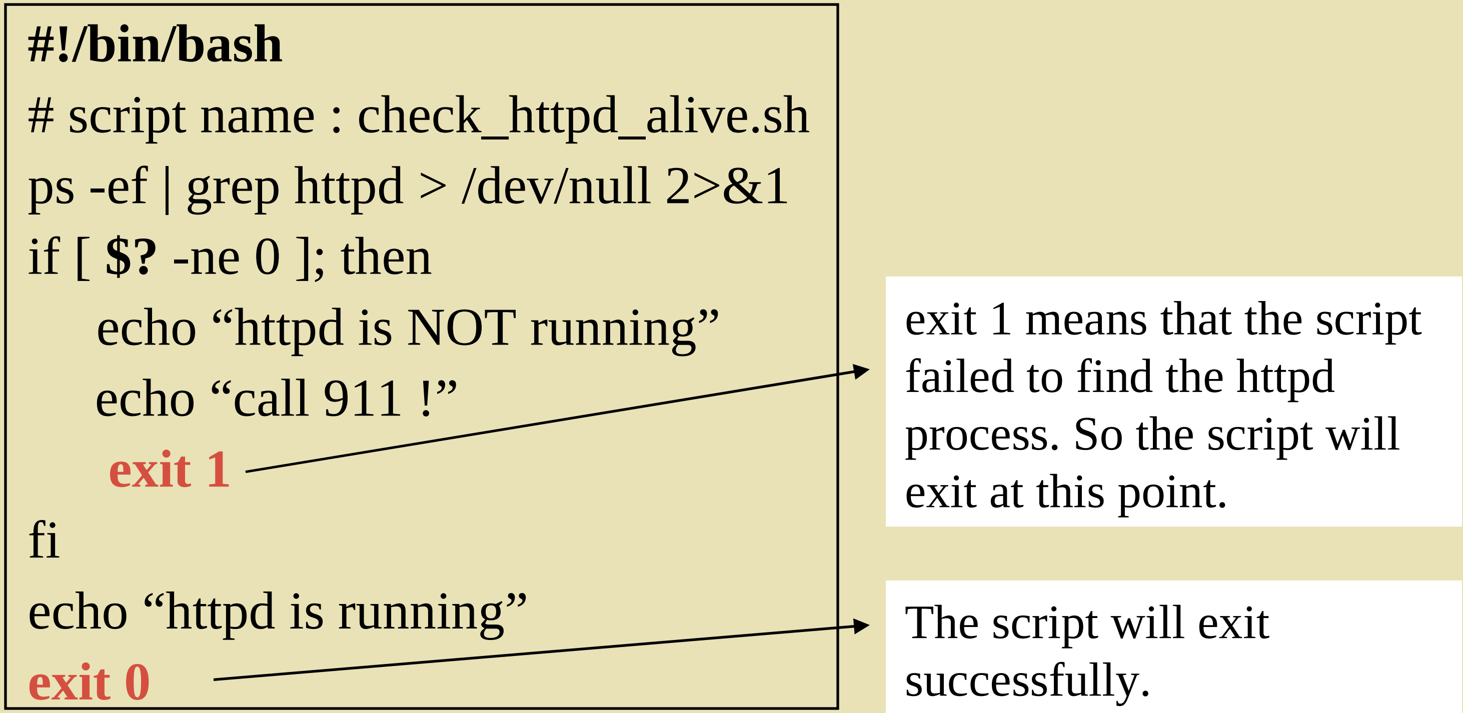
- For example:

```
#!/bin/bash
# script name : check_httpd_alive.sh
ps -ef | grep httpd |grep –v grep > /dev/null 2>&1
if [ $? -ne 0 ]; then
    echo “httpd is NOT running”
    echo “call 911 !”
else
    echo “httpd is running”
fi
```

The grep commands return 0(success) if it find httpd in the output of the ps command. If not then it return with non-zero.

Exit Code (4)

- Also your script should return an exit code which indicates if it finished OK or not.



```
#!/bin/bash
# script name : check_httpd_alive.sh
ps -ef | grep httpd > /dev/null 2>&1
if [ $? -ne 0 ]; then
    echo “httpd is NOT running”
    echo “call 911 !”
     exit 1
fi
echo “httpd is running”
exit 0
```

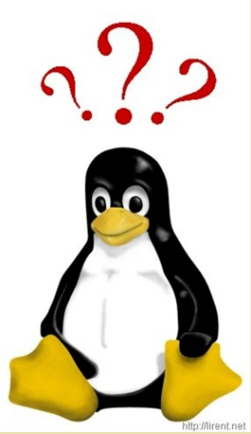
exit 1 means that the script failed to find the httpd process. So the script will exit at this point.

The script will exit successfully.

Exit Code (5)

```
#> check_httpd_alive.sh
httpd is running
#> echo $?
0

#> check_httpd_alive.sh > /dev/null 2>&1 && kill -9 `ps -eHf | grep
    httpd| grep -v grep |awk ‘{print $2}’`

#> check_httpd_alive.sh
httpd is NOT running
call 911 !
#> echo $?
1
```

Chapter 5: Outline



http://lirent.net

- What is a script?
- How to run scripts
- Variables
- Shell arithmetic's (expr)
- Controls (if, files\string\num)
- **Exercising A (systemStatusQ1,2 & checkDirSizeQ1 & Bonus Q1)**
- Controls cont… (for, while, case)
- Arguments ($0, $*, $?, $$)
- Reading input (read)
- Functions
- Traps (trap “” SIGINT)
- Debug a script (bash -x \ set -x)
- Exercising B

Chapter 5: Outline

- What is a script?
- How to run scripts
- Variables
- Shell arithmetic's (expr)
- Controls (if, files\string\num)
- Exercising A
- **Controls cont… (for, while, case)**
- Arguments ($0, $*, $?, $$)
- Reading input (read)
- Functions
- Traps (trap “” SIGINT)
- Debug a script (bash -x \ set -x)
- Exercising B

for loops

- *for* loops allow the repetition of a command for a specific set of values.
- Syntax:

```
for var in value1 value2 ... ; do
  command_set
done
```

  – command_set is executed with each value of var (value1, value2, ...) in sequence

Notes on *for* (1)

- Example: Listing all files in a directory.

```
#! /bin/bash

for i in *; do
  echo $i
done
```

NOTE: * is a wild card that stands for all files in the current directory, and *for* will go through each value in *, which is all the files and $i has the filename.

Notes on *for* (2)

- Example output:

```
# chmod u+x listfiles
# ./listfiles
a
b
c
html
listfiles
#
```

Notes on *for* (3)

- Another example: **square.sh**

```
#!/bin/bash
for i in 1 2 3 4 5; do
  echo “square of $i = `expr $i \* $i`”
done
```

=

```
#!/bin/bash
for i in `seq 5`; do
    echo “square of $i = `expr $i \* $i`”
done
```

```
#> chmod u+x square.sh
#> square.sh
square of 1 = 1
square of 2 = 4
square of 3 = 9
square of 4 = 16
square of 5 = 25
```

```
#> tmp=`seq 5`
#> echo $tmp
1 2 3 4 5
```

The *while* loop

- While loops repeat statements as long as the next UNIX command is successful.
- Works similar to the while loop in C.

Example (1)

```
#!/bin/bash
i=1
sum=0
while [ $i -le 100 ]; do
  sum=`expr $sum + $i`
  i=`expr $i + 1`
done
echo The sum is $sum.
```

NOTE: The value of i is tested in the while to see if it is less than or equal to 100.

Example (2)

```
#!/bin/bash
line_num=0
while read line; do
        line_num=`expr $line_num+1`
        echo "$ line_num: $line"
done < “/etc/hosts"
echo "Final line count is: $line_num"
```

NOTE: This script do the operation of

#> cat –n /etc/hosts

The *case* Statement

- Falls into the category of conditional statements.
- Allows the user to **branch depending** on the outcome of a string.

Notes on *case* (1)

- Syntax:

```
case string in
  pattern1)
    command_set_1
    ;;
  pattern2)
    command_set_2
    ;;
  …
esac
```

Example

```
#!/bin/bash
echo -n 'Choose option [y\n] > '
read reply
case $reply in
  “y")
      echo “the choice was y”
      ;;
  “n")
      echo “the choice was n”
      ;;
    *)
      echo Illegal choice!
      ;;
esac
```

Provide a default case when no other cases are matched.

Notes on ***case*** (2)

- We can combine two outcomes into one.
  - Using the logical **OR** in shell.
  - E.g:

```
case $reply in
  “y” | “n”)
      echo “The choice is either y or n”
      ;;
    *)
        echo “wrong choice”
        ;;
esac
```

Notes on *case* (3)

- The outcome is always checked as a string.
- The ‘;;’ are necessary to tell the shell that this option to the case is over.
- Every ***case*** statement must be terminated with an ***esac***.

Chapter 5: Outline

- What is a script?
- How to run scripts
- Variables
- Numeric operations (expr)
- Controls (if, test, files\string\num)
- Exercising A
- Controls cont… (for, while, case)
- **Arguments ($0, $*, $?, $$)**
- Reading input (read)
- Debug a script
- Functions
- Traps
- Exercising B

Command Line (1)

- Parameters to any program.
  - For example:
  - **$ ls –l foo**
  - **‘-l’** and **foo** are parameters to the program **ls**.
  - The command line for **ls** now consists of these three parameters: **ls**, **-l** and **foo**.

NOTE: command is also part of the command line

Command Line (2)

- Shell script arguments are “numbered” from left to right.
  - **$1** - first argument after command.
  - **$2** - second argument after command.
  - ... up to **$9**.
  - ${**10**} ….
  - They are called “positional parameters”.
  - E.g : #> ./CheckDirSize ~/mydir 100

$1 $2

Command Line (3)

- Example: find out if ***string*** appears in ***file.***
  - Run command as: #> ***mystr string file***

```
#! /bin/bash
grep $1 $2
```

```
#> ./mystr  root  /etc/passwd
root:x:0:0:root:/root:/bin/bash
#>
```

NOTE: $1 has value ***root*** and $2 has value ***/etc/passwd***

Command Line (4)

- Other variables related to arguments:
  - **$0** → Name of the command running.
    Tip : Usage of the script can use $0.
  - **$*** → All the arguments (even if there are more than 9).
  - **$#** → The number of arguments.
    Tip : Arguments validation in the beginning of the script.

Command Line (5)

- Example to use these special variables ./cmd_line.sh

```
#! /bin/bash
echo "$0 is the name of the command"
echo "$* is the list of arguments"
echo "$# is the total number of arguments"
#
```

Command Line (6)

- Example output:

```
#> ./cmd_line
./cmd_line is the name of the command
 is the list of arguments
0 is the total number of arguments

#< ./cmd_line 1 2 3 4 5
./cmd_line is the name of the command
1 2 3 4 5 is the list of arguments
5 is the total number of arguments
#>
```

Command Line (7)

- Another special variable (Not an argument)
  - $$ → This variable holds the PID of your current process. Example :

```
#> csh
#> echo $$
1730
#> kill -9 1730
Killed
#>
```

Reading Input

- All this while, we have talked about shell scripts that do useful work and write some output.
  - What about reading input ?
  - Done using the ***read*** command.
  - Reads one line of input and assigns it to variables given as arguments.
    - Data type of variable does not matter, as shell has no concept of data types.

Notes on *read* (1)

- Syntax:
  - **read var1 var2 var3 ….**
  - Reads a line of input from standard input.
  - Assigns first word to var1, second word to var2, ...
  - The last variable gets any excess words on the line.

Notes on *read* (2)

- Example:

```
# read var1 var2 var3
this is to test the read
# echo $var1
this
# echo $var2
is
# echo $var3
to test the read
#
```

NOTE: var3 has the rest of the string “to test the read”

Chapter 5: Outline

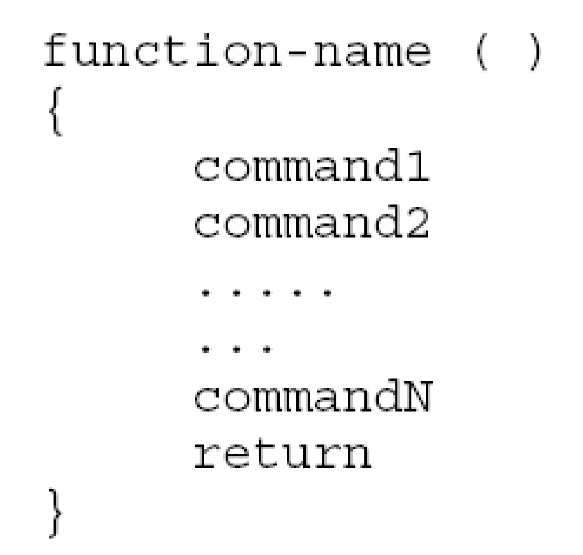
- What is a script?
- How to run scripts
- Variables
- Shell arithmetic's (expr)
- Controls (for, if, test, files\string\num)
- Exercising A
- Controls cont… (while, case)
- Arguments ($0, $*, $?, $$)
- Reading input (read)
- **Functions**
- Traps
- Debug a script
- Exercising B

Bash Functions

- Function is series of instruction/commands which **performs specific activity**.
- Function will make your code **more modular and flexible**.
- Function **reduce code duplication**.

Bash Functions

- Simple **form of function** :



```
function-name ( )
{
        command1
        command2
        .....
        ...
        commandN
        return
}
```

- To **call function** just type the **function name.**
- Function must be **define before you call** it.

Bash Script Example (1)

```
#!/bin/bash
# Description : The script count cpus or processes in the system.
usage ()
{
    abort "Usage : $0 cpus|processes"
}
abort ()
{
    echo $1
    exit 1
}
count_cpu ()
{
    if [ -f /proc/cpuinfo ]; then
        cpus_num=`cat /proc/cpuinfo  | grep "processor" |wc -l`
        echo "Number of cpus : $cpus_num"
    else
        abort "Error checking cpus"
    fi
    return
}
```

Bash Script Example (2)

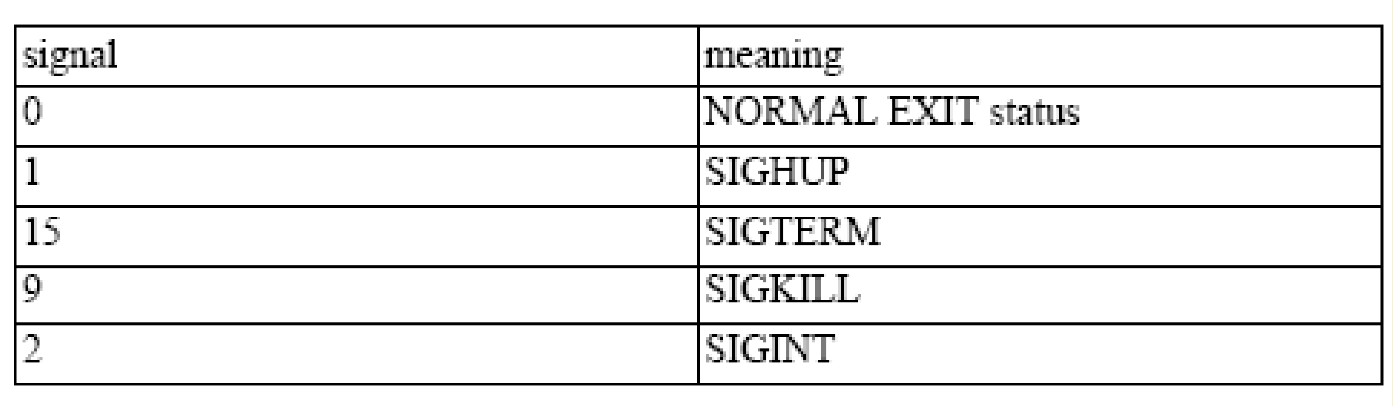
```
count_active_processes ()
{
     ps_current=`ps -ef|wc -l`
     echo "Number of processes : $ps_current"
     return
}

# ----- Main of the script -----
if [ $# -ne 1 ]; then
   usage
fi
if  [ “$1” = “cpus” ]; then
    count_cpus
elif [ “$1” = “processes” ]; then
    count_active_processes
else
    usage
fi
exit 0
```

Traps – Script Protections (1)

- Trap command will allow us to trap some or all of these signals, and perform operations on the trapped signal

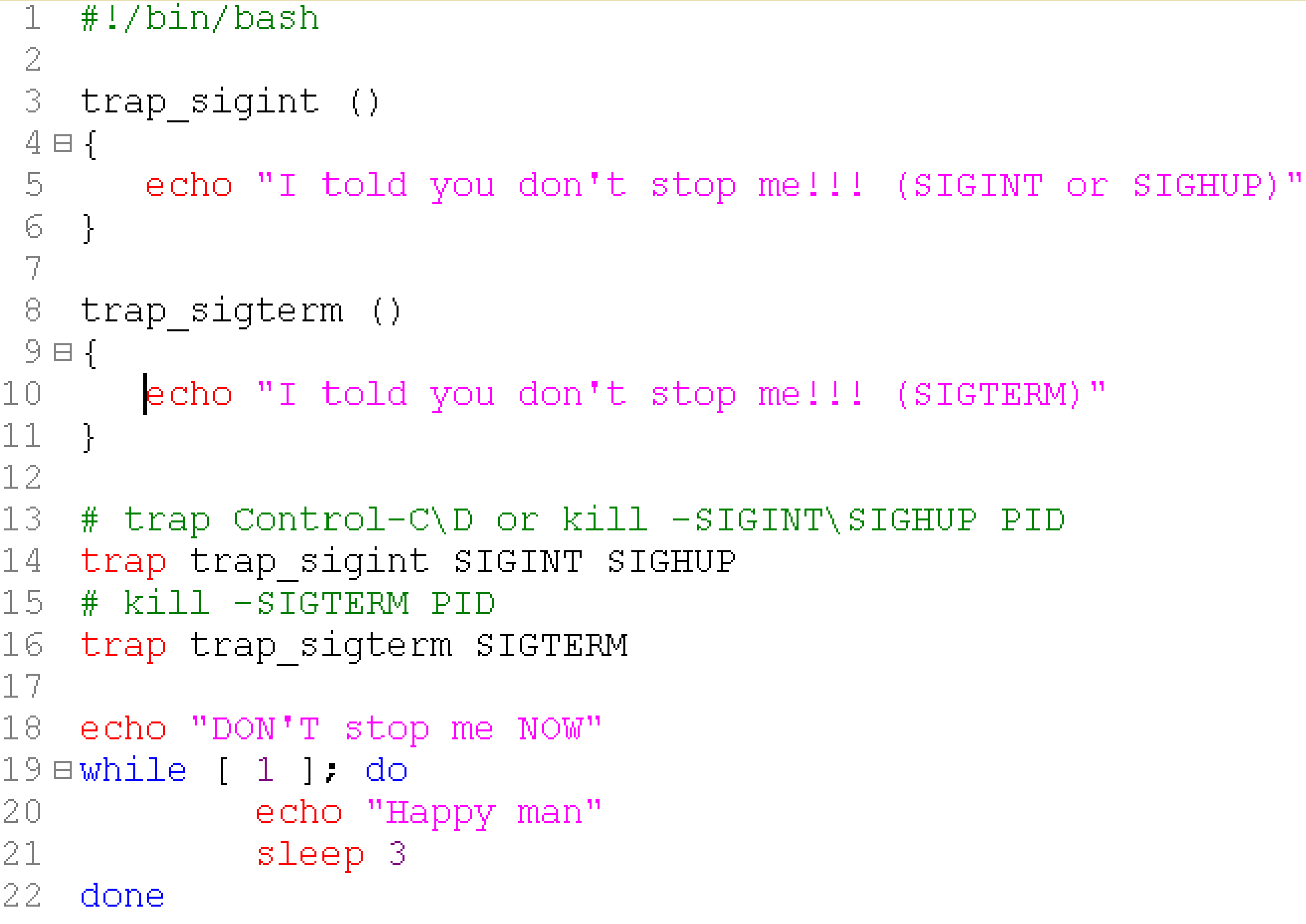
  USAGE : trap <action> <signal list>

- Potential killers out side the script



signal	meaning
0	NORMAL EXIT status
1	SIGHUP
15	SIGTERM
9	SIGKILL
2	SIGINT

Traps – Script Protections (2)



```
#!/bin/bash

trap_sigint ()
{
   echo "I told you don't stop me!!! (SIGINT or SIGHUP)"
}

trap_sigterm ()
{
   echo "I told you don't stop me!!! (SIGTERM)"
}

# trap Control-C\D or kill -SIGINT\SIGHUP PID
trap trap_sigint SIGINT SIGHUP
# kill -SIGTERM PID
trap trap_sigterm SIGTERM

echo "DON'T stop me NOW"
while [ 1 ]; do
        echo "Happy man"
        sleep 3
done
```

Traps – Script Protections (3)

- Now lets try to run it and kill it…

```
#> /tmp/dont_stop_me_now.sh
DON'T stop me NOW
Happy man
[1]+  Stopped                 /tmp/dont_stop_me_now.sh
#> bg
[1]+ /tmp/dont_stop_me_now.sh &
#> Happy man


#> kill -SIGINT `ps -ef| grep dont_stop| grep -v grep |awk '{print $2}'`
I told you don't stop me!!! (SIGINT or SIGHUP)
Happy man
Happy man
```

Traps – Script Protections (4)

- Continue to kill

```
#> kill -SIGHUP `ps -ef| grep dont_stop| grep -v grep |awk '{print $2}'`
I told you don't stop me!!! (SIGINT or SIGHUP)
Happy man
Happy man


#> kill -SIGTERM `ps -ef| grep dont_stop| grep -v grep |awk '{print $2}'`
I told you don't stop me!!! (SIGTERM)
Happy man
Happy man


#> kill -9 `ps -ef| grep dont_stop| grep -v grep |awk '{print $2}'`
[1]+  Killed                  /tmp/dont_stop_me_now.sh
```

Debug a Script (1)

- A **simple way to debug** a script is **by** adding **echo commands** inside it.

  Echo commands will help you to trace the script’s progress and **catch the bug**.

Debug a Script (2)

- **Bash debugging feature:**
  Use the **–x** in the bash command.
  Script for example:

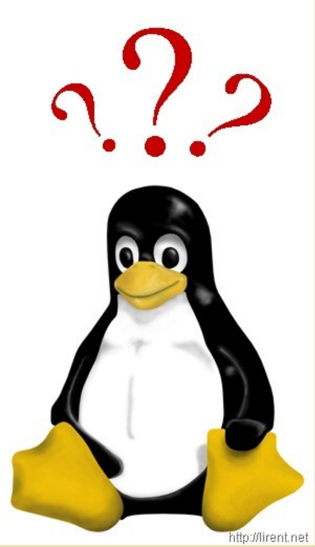
```
# cat > /tmp/debug
tot=`expr $1 + $2`
echo $tot
^D
```

```
# bash /tmp/debug 4 5
9
```

```
# bash -x /tmp/debug 4 5
++ expr 4 + 5
+ tot=9
+ echo 9
9
```

Chapter 5: Outline

- What is a script?
- How to run scripts
- Variables
- Shell arithmetic's (expr)
- Controls (for, if, test, files\string\num)
- Exercising A
- Controls cont… (while, case)
- Arguments ($0, $*, $?, $$)
- Reading input (read)
- Functions
- Traps
- Debug a script
- **Exercising B (systemStatusQ3 & checkDiskSizeQ2,3,4 & psMon)**



Chapter 6 :
Useful Commands

Chapter 6: Outline

- find
- file
- s\diff
- screen
- alias
- script
- Exercising

find

- Finding files in your system.
- Examples:

```
#> find . -name "*.pdf"             =>   "find ." = search from current dir
#> find . -name "[0-9]*"            =>    find files\dirs starting with a number
#> find . –type l                   =>   "find ." = find only symbolic link
#> find . -type d -name "[0-9]*"  =>  find only dirs which?
```

***Delete** all the files in /tmp which were not **modified 7 days ago**:*

```
#> find . -mtime +7 -exec rm -r >/dev/null 2>&1 {} \;
```

Same but…

```
#> find . -mtime +7 | xargs rm -r >/dev/null 2>&1
```

file

- Find out the file type
- Examples:

```
 #> file    /etc   /etc/inittab   /sbin/shutdown    /bin/pwd
/etc:                          directory
/etc/inittab:                  ASCII English text
/sbin/shutdown:                symbolic link to `../lib/upstart/shutdown'
/bin/pwd:                      ELF 32-bit LSB executable....
```

diff (1)

- File comparing
- Without any options, it produces a series of lines containing Add (a), Delete (d), and Change (c) instructions
- The sdiff utility is similar to diff but display the output in a side-by-side format
- Usage:

```
diff [options] file1 file2
diff [options] file1 directory
diff [options] directory file2
diff [options] directory1 directory2
```

diff (2)

- Example:

```
#> cat /tmp/filea
a
b
c
```

```
#> cat /tmp/fileb
a
X
c
```

```
#> diff /tmp/filea /tmp/fileb
2c2
< b
---
> X
```

```
#> sdiff /tmp/filea /tmp/fileb
a                              a
b                            | X
c                              c
```

screen

- Two uses can **share the same terminal!**
- Useful for solving problem together.
- How to use it:

Person1

1. Log in to serverA with username.
2. Execute the command: **screen -S user1term**.
3. Wait for other users to contact my “user1term”

Person2

1.Log in to serverA with the same username.

2. Execute the command: screen **-x user1term**.
3. Now both of the users can share the same terminal

Alias (1)

- Aliases:
  - A substitution of one symbol for another.
  - Time saving tool by reducing key strokes for “common” commands.
  - Lifetime is the current session unless created in configuration files.

Alias (2)

- Creating:

  Usage : alias aliasname=‘command’

  ```
  #> alias ls=‘ls –l’
  #> alias rm=‘rm –i’
  #> alias h=‘history’
  #> alias ..=‘cd ..’
  ```

- Deleting:

  Usage : unalias aliasname

  ```
  #> unalias ls
  ```

script Command (1)

- The *script* command is used to make a record of an interactive session.
  - **script [-a] [file]**
  - -a option is used to record actions to file by appending contents to file.
  - When *script* command is started, it starts a new *shell* and once you type ***exit***, you return to your original shell.

script Command (2)

```
#> script /var/tmp/file.log
Script started, file is test
erase ^? intr ^C kill ^U
#> pwd
/afs/cs.pitt.edu/usr0/axgopala
#> exit
exit
Script done, file is test
#>
```

Note: All the typing saved into the /var/tmp/file.log

Chapter 6: Outline

- find
- file
- s\diff
- screen
- alias
- script
- **Exercising**

Appendix:
Enhanced Commands

Appendix: Outline

- rpm query
- netstat
- lsof
- chkconfig
- nslookup
- wget
- dos2unix
- crontab

rpm -q

- Is a query command for rpms,
- Examples:
- rpm -qa // show all installed rpms
- rpm -q <rpm> // show if rpm installed
- rpm -ql <rpm>// list the files in the installed rpm
- rpm -qf <file> // show the rpm that brouth the file
- rpm –qi <rpm> // show details of the installed rpm
- rpm –q --scripts <rpm> // show the pre\post scripts
- rpm2cpio <rpm> | cpio -ivd // extract files from the rpm(not installing them)
- rpm -q kernel --qf '%{NAME} - %{VERSION} - %{RELEASE} . %{ARCH}\n‘
  e.g: *kernel - 2.6.18 - 53.1.13.el5 . i686*

netstat

- Shows connections, routing information, statistics
- Possible uses:
  - find systems that your system has recently talked to, find recently used ports
- Examples:

```
netstat –rn     // routing tables
netstat     // open sockets, etc.
netstat –s // summary statistics
netstat –p // programs
```

lsof

- Lists open files on your system
- Useful to see what processes are working with what files and ports
- Usage: lsof

```
#> lsof –c bash      // List all files by processes name
#> lsof -i tcp:ssh   // What process has the port
#> lsof -i tcp:22
  Note : /etc/services
```

Linux Run-Levels

0	Halt
1	Single user mode
2	Multiuser, w/o NFS
3	**Full multiuser mode**
4	unused
5	X11
6	reboot

Chkconfig (1)

- How can you control what programs will start up and shut down and find out what is actually running?
- The chkconfig command is used to activate and deactivate services.
- #> chkconfig --list:

  list of the services and their run-level activation.

- Example :

```
#> chkconfig  --list | egrep "crond|nfs"
crond        0:off   1:off   2:on    3:on    4:on    5:on    6:off
nfs          0:off   1:off   2:off   3:off   4:off   5:off   6:off
nfslock      0:off   1:off   2:off   3:on    4:on    5:on    6:off
```

chkconfig (2)

- Examples:

  - List all the services and their init levels:

  ```
  #> chkconfig --list
  ```

  - Forces httpd service to run for levels 3,4,5.

  ```
  #> chkconfig --level 345 httpd on
  ```

  - Disable the httpd service to run at all.

  ```
  #> chkconfig httpd off
  ```

nslookup

- Potential Uses:
  - Query internet name servers
  - Find name for IP address, and vice versa

- Usage:
  - **nslookup** <ip> | <hostname>
    - E.g. nslookup data.cs.uwec.edu

wget

- The non-interactive network downloader (http\s,ftp)

  - GNU Wget is a free utility for non-interactive download of files from the Web.

- Usage:
  - **wget** <url>

dos2unix

- DOS/MAC to UNIX text file **format converter**.
- **When to use it?**

  If you create a text file in DOS it has **other line endings** than if you create it on a Linux/UNIX system. Where DOS appends a RETURN and a NEW LINE at the end, Linux just uses the NEW LINE character. You use dos2unix to convert these to the Linux format.
- **What happened** when you trying to run script from dos in Unix -> you may get the error : **“Bad interpreter”.**
- **How to use it?**

  ```
  #> dos2unix dos-script-file  > unix-script-file
  ```

  Now you can run safely the script unix-script-file.

cron – deamon (1)

- UNIX’s answer to automated job scheduling.
- Used to schedule jobs to run at particular time or at a particular frequency.
- Useful to automate system administration tasks.
- Is actually a background system process `-crond` (the cron daemon) is started at boot time from rc scripts.

crontab – files (2)

- Configuration files defining scheduled jobs are stored in multiple locations on some systems:
  - `/var/spool/cron`

    The mail Cron directory, where jobs defined according to username. These definition files are called *crontab* files.

crontab - Format File (3)

- Each line in crontab has five fields:
  - Minute - (0-59)
  - Hour - (0-23)
  - Day of the month - (1-31)
  - Month of the year - (1-12)
  - Day of the week - (0-6) (Sunday is 0)
  - Command line - executed command
- Example:

```
0 8 * * 2      echo “Happy Monday Morning”
* * * * *      echo “One Minute Passed > /tmp/log
0 1,2,3 1 1 0    /bin/cleanOldFiles /tmp
```

crontab – command (4)

- View crontab information: crontab –l
- Editing crontab file: crontab -e

Appendix: Outline

- rpm query
- netstat
- lsof
- chkconfig
- nslookup
- wget
- dos2unix
- crontab

References – Linux shell scripting

- http://learnlinux.tsf.org.za/courses/build/shell-scripting/shell-scripting.pdf
- http://www.dis.uniroma1.it/~bordino/shell-tutorial.pdf
- http://rapidshare.com/files/178929213/Linux_in_a_Nutshell__3rd_Ed.___O_Reilly_.pdf
- http://www.usinglinux.org/docu/guides/linuxcookbook-1.2.pdf
- ftp://211.68.71.80/pub/Documents/Programming/Shell/Advanced%20Bash%20Shell%20Scripting%20Guide.pdf

The END.

Written by :
Shay Berman

Based on lectures in
http://www.cs.pitt.edu/~axgopala/cs132/notes/lecture24.ppt