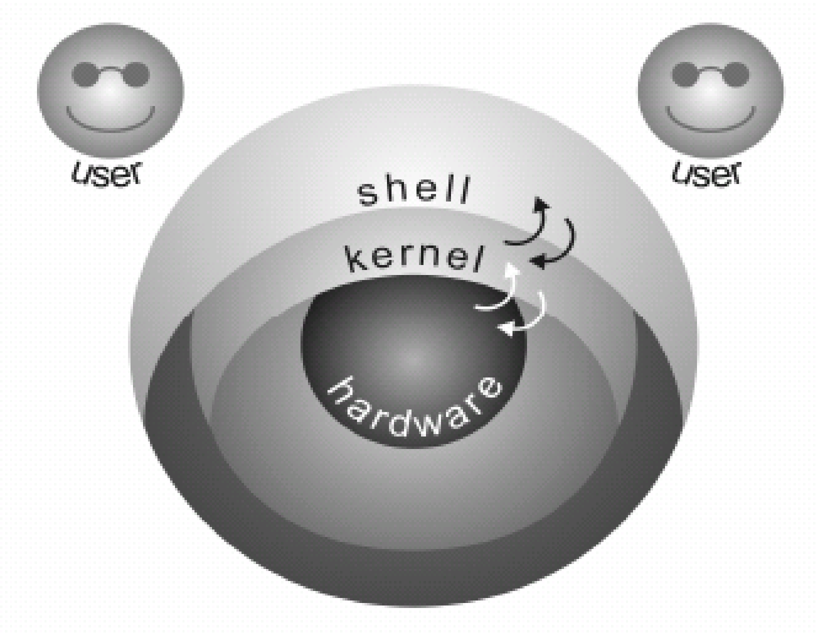
Linux/Unix Shell Scripting

A Brief Introduction

What is shell ?

- User talks to shell.
- Shell talks to kernel.
- Kernel talks to hardware.
- Hardware does the job.



Shell (cont.)

- You can write commands or write shell scripts.
- Shell programs are called shell scripts.
- Many Unix administrative programs are written as shell scripts.
- A Unix shell is actually an interpreter of a programming language.

Shell script

- Script is a text file that has:
  - Shell commands.
  - Control structures.
- There are many shells available:
    - Bash
    - Sh
    - Csh
- To see a full list of your valid shells:
  - `$cat /etc/shells`

Command processing

- There are alternatives:
  - Run command itself: echo &variable.

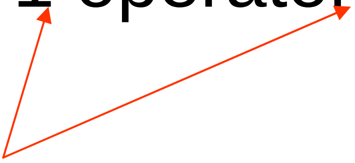
    User ←→ shell
  - Call child shell: ls -l

    User ←→ shell ←→ child shell
  - Call kernel: cpio

    User ←→ shell ←→ kernel

Some useful commands

- expr
  - Usage :expr integer1 operator integer2



Separated by space

- Sum
- Subtract
- Multiply
- Divide
- Reminder

– expr 10 + 2

– expr 6 * 4

Useful commands (cont.)

- alias
  - – Usage: alias name=‘value’
- Usually define aliases in
  - ~/.bashrc
  - ~/.bash_profile
- Use semicolon for sequence of commands

- alias → show all aliases
- unalias alias_name

- $ alias deltree=‘rm –rf’
- $ alias ls=‘ls –l | more’
- $ deltree
- $ \ls

Useful commands (cont.)

- Shell stores previous used commands in a file:
  - ~/.bash_history
- Can use history to see them.
- Its size is limited: $HISTSIZE
- Its name is in $HISTFILE

Why we use shell scripts

- Manipulating system scripts.
  - start up and shutdown scripts
- Refrain from doing redundant jobs.
  - Write a script one time and use it.
- Mechanization of hard works.
  - Don’t use compounded arguments each time.

How to write scripts

- Use an editor:
  - vim, emacs, pico
- Determine the path and name of interpreter:

  #!/bin/bash
- Can use $echo $SHELL if you don’t know it.

How to write (cont.)

```
$vim first.sh →
```

Is not mandatory

```
#!/bin/bash
echo “hello world”
echo –e “here is: \c”
pwd
```

```
$./first.sh
bash: ./first.sh: permission denied
$chmod 755 first.sh
$./first.sh
hello world
here is: /home/root
```

How to write (cont.)

- Run script explicitly:

  ```
  $sh first.sh
  ```

  - In this mode you haven’t to set permissions.

Variables

- They haven’t any type in shell.
- Name of variables:
  - Started with a..z, A..Z, _.
  - Can have a..z, A..Z, 0..9.
- Variable definition:

```
$ name=value
```

There is not space next to =.

Variables (cont.)

- To access to variables value, must use $.

```
$ myvar=/home/ssclinux
$ echo $myvar
/home/ssclinux
```

- Value can’t have space.
  - Solution:

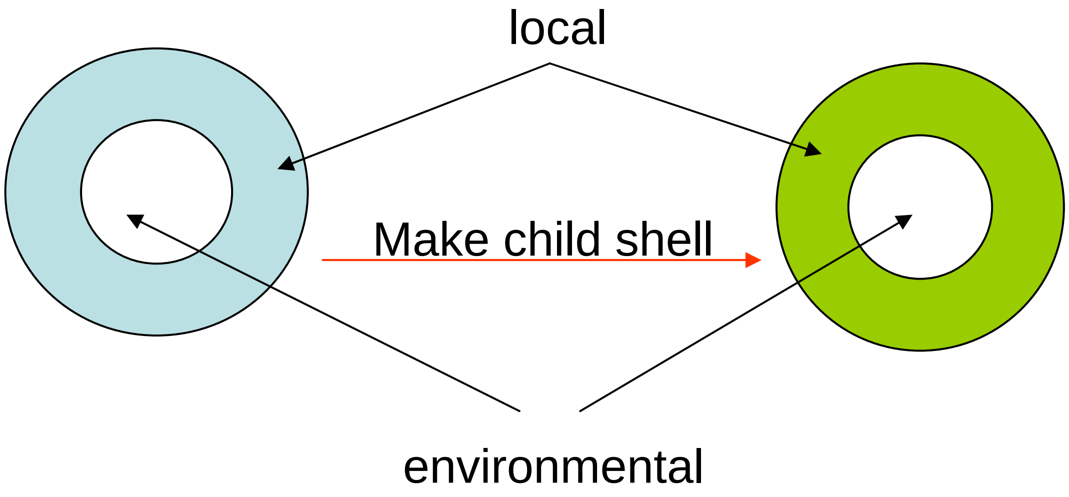
    ```
    myvar=“I have space”
    ```

Variables (cont.)

- Also shell has arrays.
    - learn it by yourself ☺
- Variables:
    - Local
        - Defined by user.
        - Only accessible in current shell.
    - Environmental
        - Accessible in current and child shell.
        - Use printenv to see them
        - Most of them get their value on login time.

Variables (cont.)

- Can use export to make environmental variables:
    - $ export myvar
- Use env to see all exported variables.



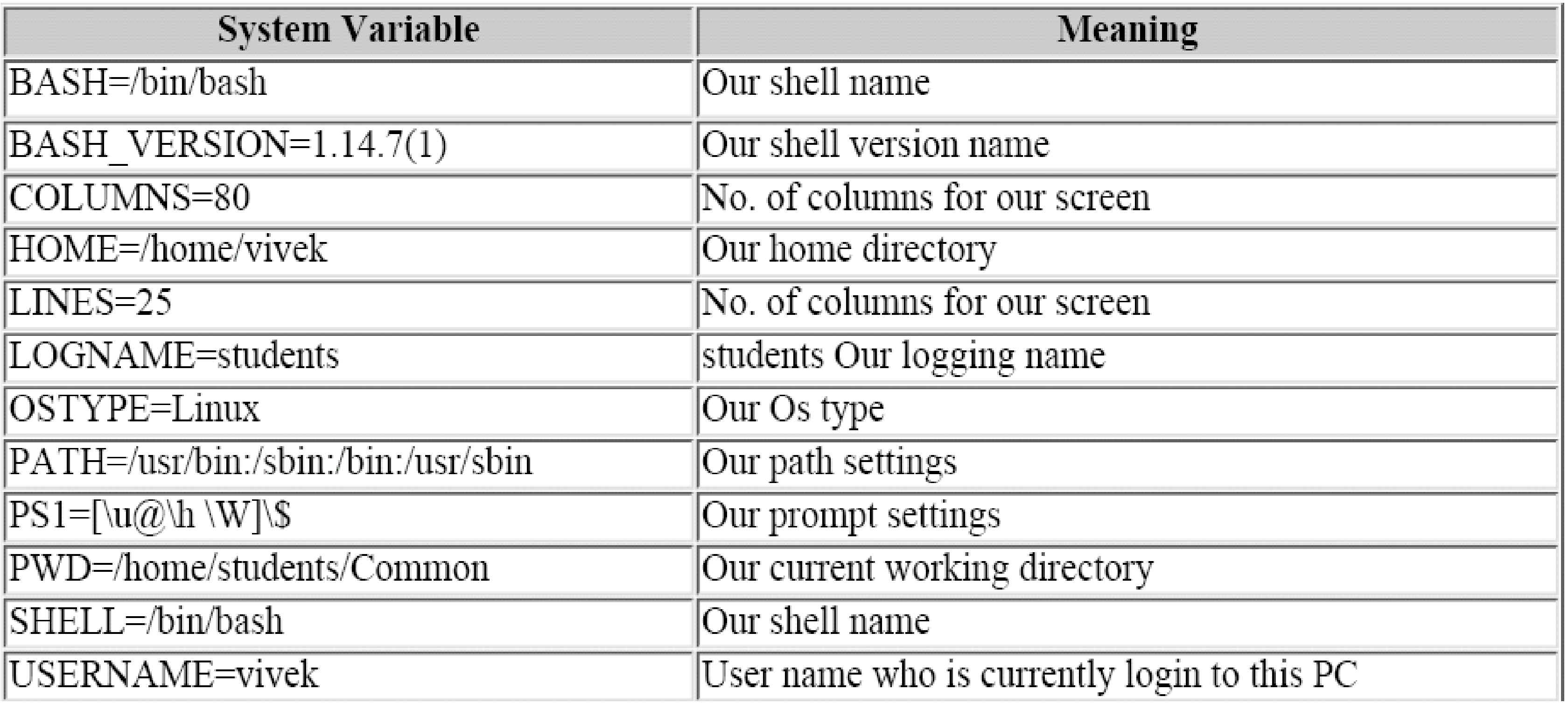
Variables (cont.)

```
$ myname=Ali
$ echo $myname
Ali
$ bash          ←――――――――――――  Make child shell
$ echo $myname

               ←――――――――――――  nothing
$ exit
```

Variables (cont.)

- Some important variables:



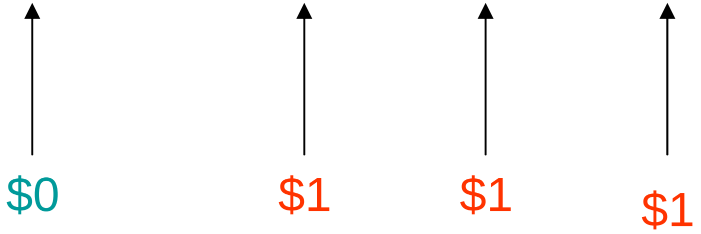
System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/vivek	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	students Our logging name
OSTYPE=Linux	Our Os type
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=vivek	User name who is currently login to this PC

Variables (cont.)

```
$ vim setperm →
```

```
#! /bin/bash
chmod u+x “$@”
exit 0
```

```
$ ./setperm file1 file2 file3
```



$0 $1 $1 $1

$# = number of arguments

Input variables

- Use read command:
  - Usage: read var1 var2 var3 …
- Read one line from input and put words in variables.
- Read can do more
  - See it yourself ☺
- Also see shift and set.

Input and Output

- 0→ stdin: shell use this descriptor to get input.
    - Keyboard is default.
- 1→ stdout: use this for output.
    - Screen is default.
- 2→stderr: use this for error.
    - Screen is default.

Redirection

- Use redirection meta characters to change defaults:

>	redirect standard output
<	redirect standard input
>&	redirect standard output and error
|	redirect one’s output to other’s input
>>	append to standard output
>>&	append to standard output and error
<<	here documents. ← use for input.

- Also we can use 0>, 1>, 2> .

Redirection (cont.)

- Using temporarily files:

```
$ who > tempfile
$ sort < tempfile > sortedfile
$ lpr sortedfile
$ rm –f sortedfile tempfile
```

- Using /dev/null to eliminate output and error

```
$ cat file.text > /dev/null
```

Piping

- Or use piping:

```
$ who | sort | lpr
```

- Pipes have more speed because they run simultaneously.
- They are one way.

Control flow structures
if

- if command:
  - Usage:

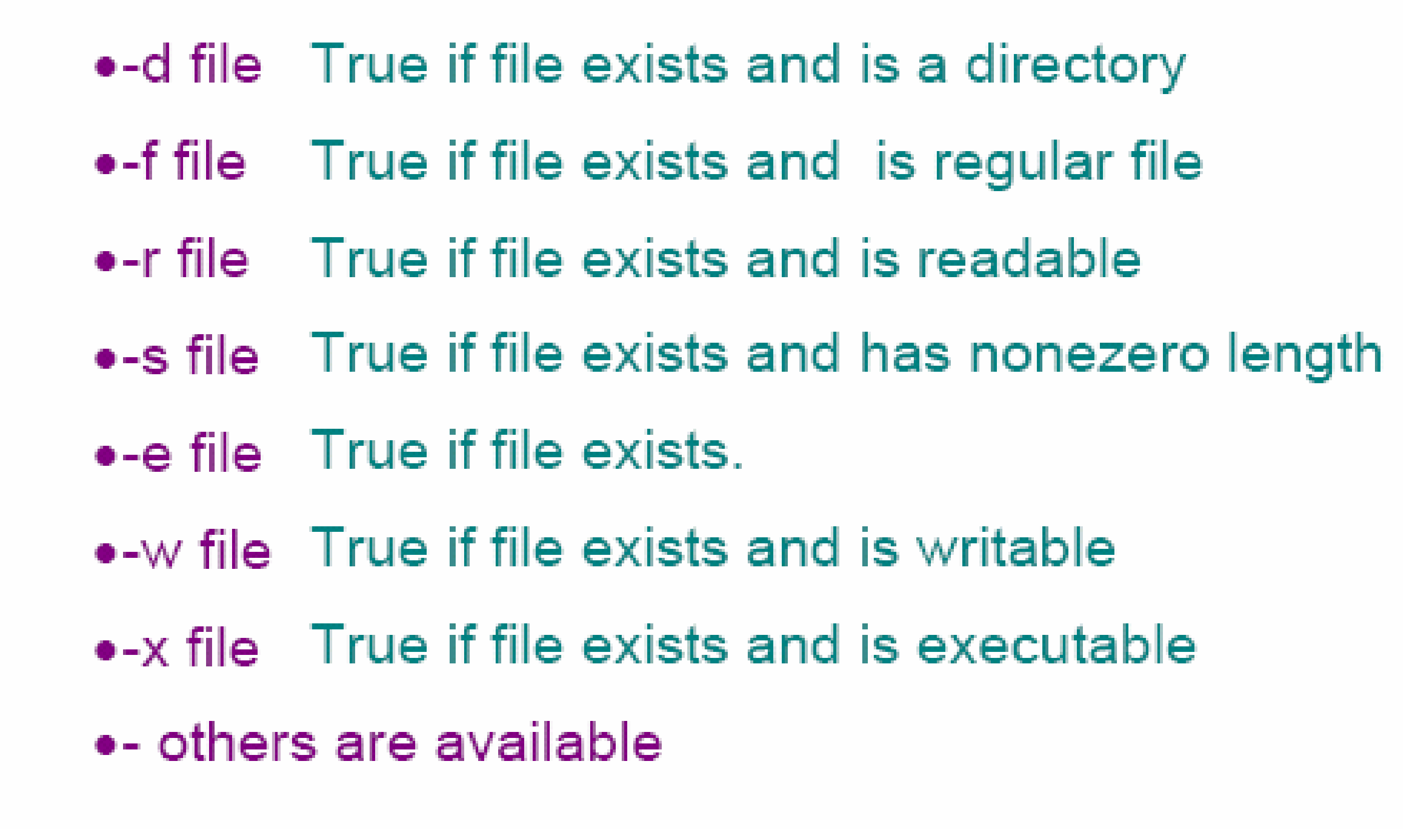
```
if command
then
        command 1
        command 2
        …
fi
```

Control flow structures
if (cont.)

- Use test for condition checking.
  - Evaluate an expression and return 0 if it was true else return a number (not zero).
  - Usage: test expression

    OR: [expression]
  - Three types:
    1. File tests
    2. String comparisons
    3. Numerical comparisons

File tests



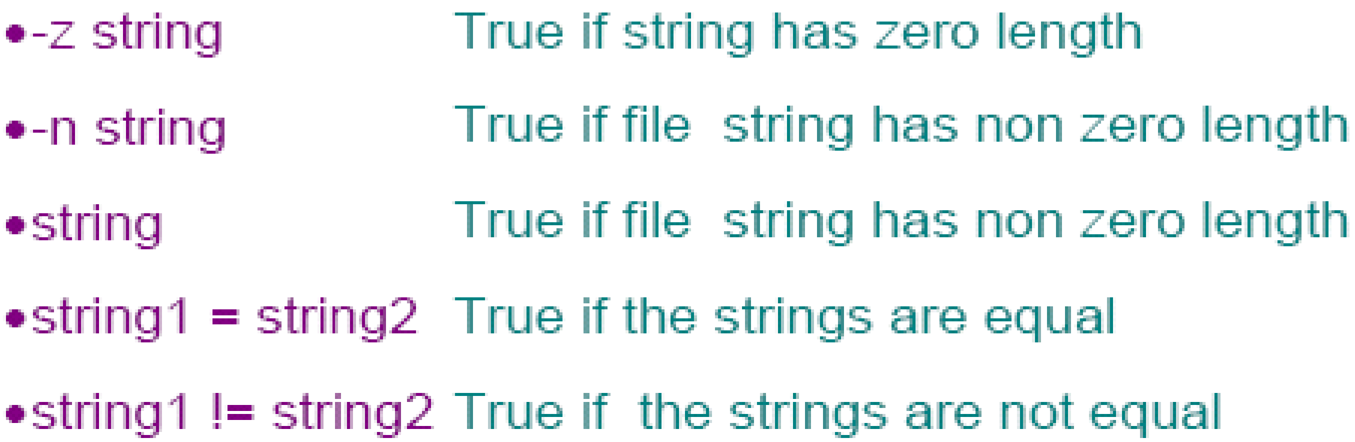
- -d file True if file exists and is a directory
- -f file True if file exists and is regular file
- -r file True if file exists and is readable
- -s file True if file exists and has nonezero length
- -e file True if file exists.
- -w file True if file exists and is writable
- -x file True if file exists and is executable
- - others are available

File tests (cont.)

```
$ touch file.txt
$ test -f file.txt
$ echo $?
0
$ [ -d file.txt ]
$ echo $?
1
```

Shows the last command execution return.

String comparisons

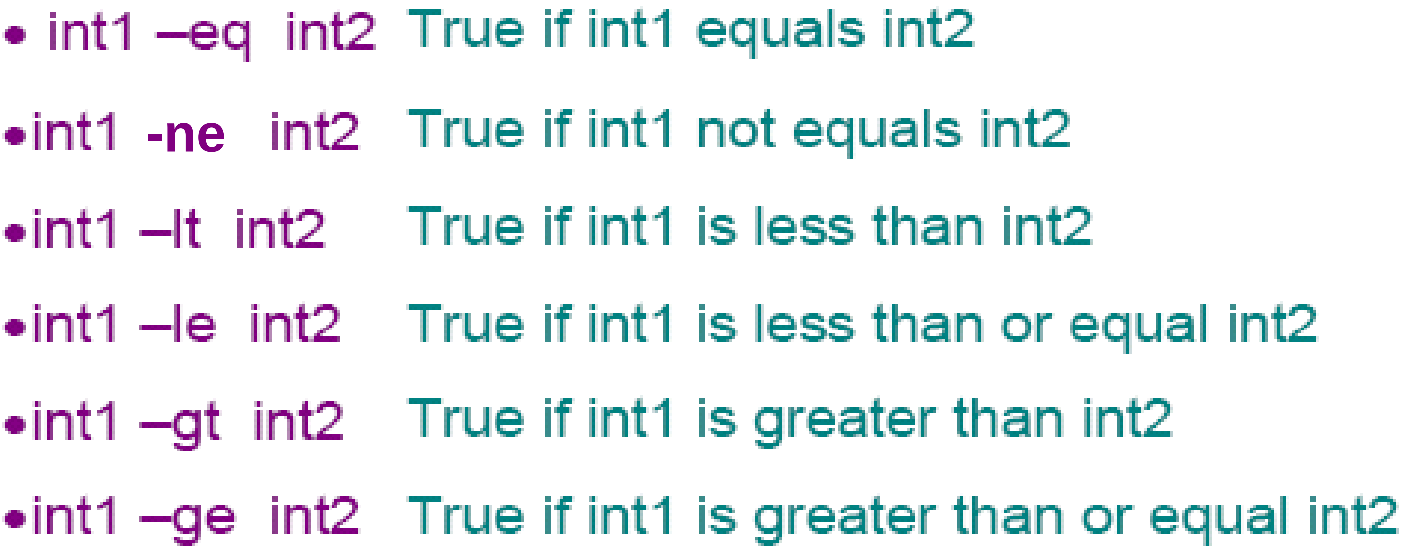


-z string	True if string has zero length
-n string	True if file string has non zero length
string	True if file string has non zero length
string1 = string2	True if the strings are equal
string1 != string2	True if the strings are not equal

String comparisons

```
#!/bin/bash
x=“salam”
if [ “$x” = “salam” ]
   echo “condition was true”
fi
```

Numerical comparisons



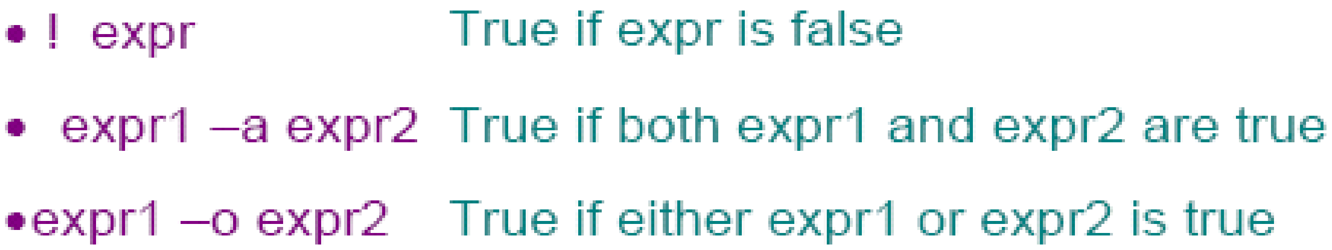
int1 –eq int2	True if int1 equals int2
int1 **-ne** int2	True if int1 not equals int2
int1 –lt int2	True if int1 is less than int2
int1 –le int2	True if int1 is less than or equal int2
int1 –gt int2	True if int1 is greater than int2
int1 –ge int2	True if int1 is greater than or equal int2

Numerical comparisons

Used for commands.

```
#!/bin/bash
if test ‘who | wc –l’ –ge 1
then
   echo “it is not safe to shutdown”
else
   echo “now, it is safe to shut down”
fi
```

Other test operators



- ! expr True if expr is false
- expr1 –a expr2 True if both expr1 and expr2 are true
- expr1 –o expr2 True if either expr1 or expr2 is true

- expr is a valid test command.

More example

```
If [ “$UID” –ne “ROOT_ID” ]; then
   echo “you must be root”;
   exit 1;
fi
```

Case

- Sample usage:

```
read arg
case “$arg” in
   a) echo “the arg is a” ;;
   b) echo “the arg is b” ;;
   ..
   *) echo “can not recognize arg”;;
esac
```

While

- Sample usage:

```
ANSWER=
while [ -z “$ANSWER ]
do
    echo “Enter the name of a directory \
            where files are located”
    read ANSWER
    if [ ! -d “$ANSWER” ]
            echo “error: invalid directory name”
            ANSWER=
    fi
Done
exit 0
```

A practical example

```
#! /bin/bash
PERIOD=900
currentline=‘cat /var/log/messages | wc –l’
while true
do
        echo “press CTRL+C to terminate”
        sleep $PERIOD
        newline=‘cat /var/log/messages | wc –l’
        dif=‘expr ${currentline} - ${newline}’
        diflines=‘cat /var/log/messages | tail ${dif}’
        echo “$diflines” | mail –s “updatelog” root
        currentline=“$newline”
done
```

Until

- Usage:

```
until command
do
    command1
    command2
    …
done
```

  - Use it yourself ☺

for

- Usage

```
for var in word_list
do
    commands.
    …
done
```

for (cont.)

```
for file in ‘ls’
do
   if [ –d “$file” ]
        rm –rf “$file”
   fi
done
```

for (cont.)

- Another usage:

```
for (( expr1; expr2; expr3 ))
do
        ...
done
```

- Example:

```
for (( i = 0 ; i <= 5; i++ ))
do
        echo "Welcome $i times"
done
```

select

- Useful for making menus.
- See it yourself ☺

Regular expressions

- reg exp is a simple description of a pattern.
- Used in many utilities: awk, grep, sed.
  - $ grep ali /etc/passwd
  - $ grep [Aa]li /etc/passwd
- They differ in different utilities.
- It is better to put them in single quote.

Regular expressions meta characters

- Dot .
  - Compatible with a single character
    - a.b = { asb, acb, a2b, a$b }
    - a.b != { ab, assb, a$bbb }
      - Newline is an exempt.
- Star *
  - Compatible with occurrence of last character for zero or more times.
    - a*b = { b, ab, aab }

Regular expressions meta characters (cont.)

- ^
  - Compatible with start of line.
    - ‘^ali’ = every line that starts with ali.
- $
  - Compatible with end of line.
    - ‘end$’ = every line that ends with end.
- [..]
  - Compatible with the set of chars determined in braces.
  - [aeiou] = all of vowel chars.
  - [a-d] = {a, b, c, d}

Regular expressions meta characters (cont.)

- [^..]
  - Compatible with every chars except those determined in the set.
- [:alpha:]
  - Compatible with asci chars: a-z, A-Z
- [:digit:]
  - Compatibe with digits: 0-9
- [:alnum:]
  - Compatible with digits and asci chars.

Regular expressions meta characters (cont.)

- [:lower:]

  → lower case chars.
- [:upper:]

  → upper case chars.
- [:space:]

  → {space, tab, newline, carriage return, vertical tab}
- [:xdigit:]

  → Numbers in Hex.
- [:punct:]

  →{ !, #, “, %, &,’ , (,), \, ;, <, =, >, ?, [,], *, +, , , - , ., /, :, ^, _, {, |, } }
- [:graph:]

  → { alnum, punct }

Regular expressions meta characters (cont.)

- \+
  - Compatible with occurrence of last character for one or more times.
    - a\+b = { ab, aab, aaab }
- \?
  - Compatible with occurrence of last character for zero or one time.
    - a\?b = { ab, aab, aaab }
- \|
    - a\|b = { a, b }

Regular expressions meta characters

- \{N\}
  - – Occurrence of last char for N times.
- \{N,M\}
  - – Occurrence of last char for at least N and maximum of M times.
    - [a-z]\{3,10\} = all of lower case strings with length of 3 to 10.
- \{N,\}
  - – Occurrence of last char for at least N times.

Regular expressions meta characters (cont.)

- \>
  - – End of word.
    - ‘ix\>’ = words end with ix.
- \<
  - – Start of word.
    - ‘\<un’ = words start with un.
- \(chars\)
  - – To memorize part of regular expression.
    - ‘atten\(tion\|dant\)s’ = { attentions, attendants }

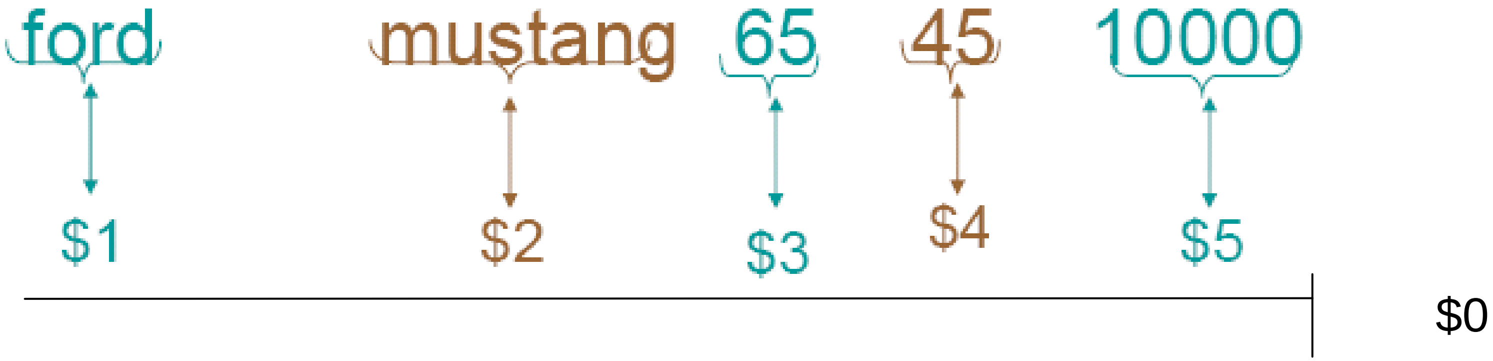
Grep

- Searching for a pattern in a file.
- Usage:
  - grep [option] regexp file1 {file2, …}
    - Without option: output the lines with pattern.
    - -v : output the lines without pattern.
    - -c : output number of lines that have pattern.
    - … see yourself ☺
  - lastlog | grep –v root
  - dmesg | grep isa

AWK

- Designed by
  - Alfred v.Aho
  - Peter j.Weinberger
  - Brian w.Kernigan
- Used for data processing and report producing.
- Searches the input file far specific pattern , then does the action.

AWK (cont.)



- Every line is a record.
- Every line has at least a field.
- Field separator is <tab> by default.
- Field named like: $1, $2.
- $0 is a name for all of line.

AWK (cont.)

- Different modes of use:
  - – Awk ‘program’ input_files
  - – Awk ‘program’
  - – Awk –f program_file input_files

AWK (cont.)

- Simple usage:
  - Awk {action} pattern {action} …pattern action

- $ ls –l | awk ‘/^d/ {print “rm –r ”$9 }’ | bash
- $ ls –l | grep –v ‘^d’ |

  awk ‘{print “rm –f ”$9 }’ | bash

AWK (cont.)

```
BEGIN{action}              #optional
{action}
pattern {action}
.
.
pattern {action}
END{action}                #optional
```

AWK (cont.)

- vim countfld.awk →

```
#! /bin/awk -f
BEGIN {filecount = 0 ; dircount = 0}
/^-/ {filecount = filecount +1}
/^d/ {dircount = dircount +1}
END { print "\n"
   print "Total number of files: " filecount
   print "Total number of directories: " dircount}
```

```
$ ls –l | awk –f countfld.awk
```

sed

- the Stream EDitor.
- Usually used for editing files.
- Is not user-friendly!
- Learn it yourself☺.

☺Thank you☺