

Introduction to Cluster Computing:

Linux, shell scripting, queuing systems, cluster architecture

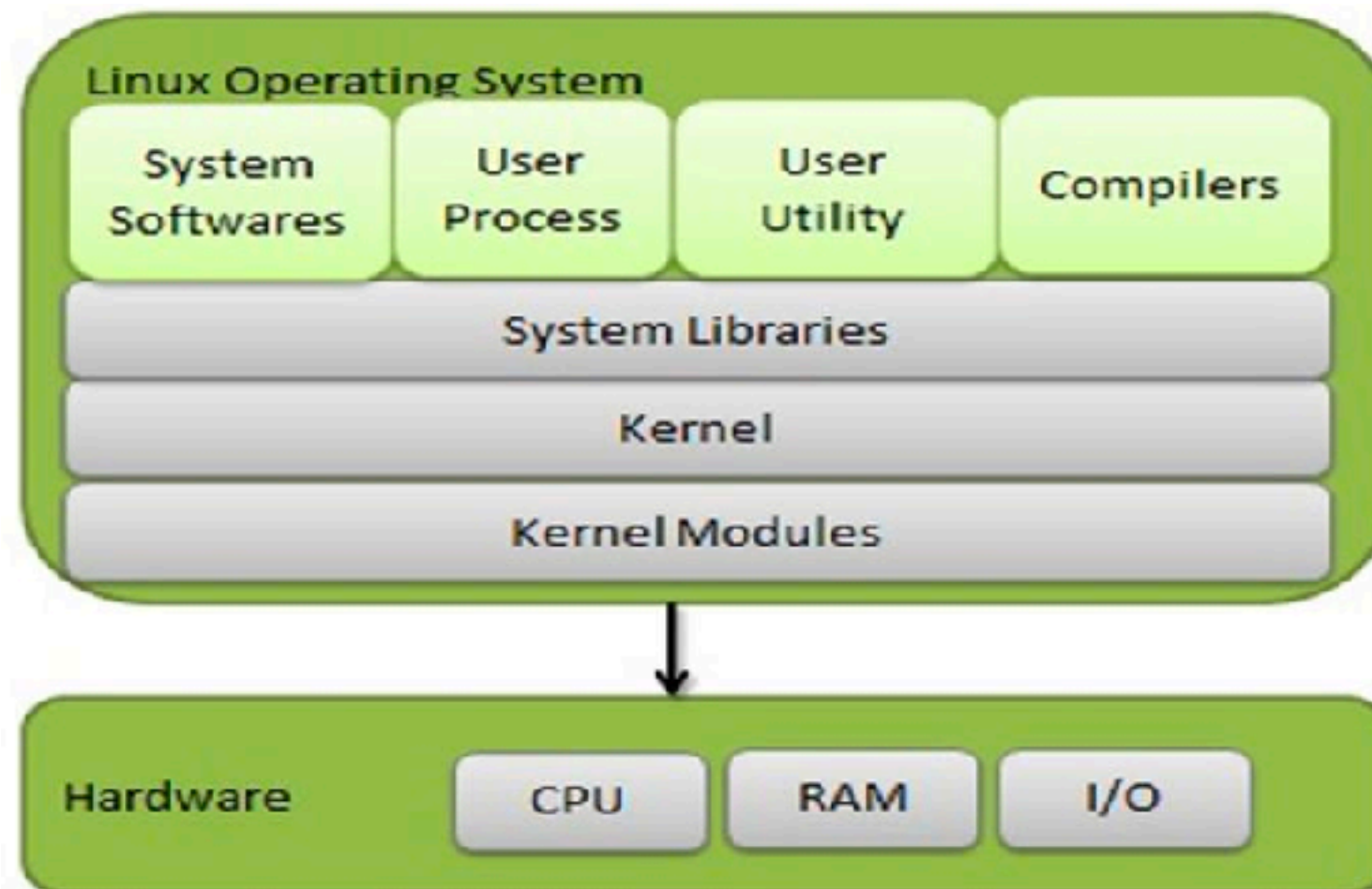


Instructor: Dr. Peggy Lindner (plindner@uh.edu)

Lecture 2

Linux

- is an OS just like Windows or Mac OS X
- is the kernel: the program in a system that allocates the computer/server hardware resources to the other programs
- normally used in combination with the GNU operating system utilities: the whole system is basically GNU with Linux added, or GNU/Linux



Linux



- General features of Linux:
 - Most distributions are free
 - Open-source (completely customizable)
- Portable to nearly any hardware platform (cell phones, roku, steamOS devices, PS3, tablets, TVs, routers)
- Highly scalable to lots of cores, and or lots of memory
- Robust and proven security model
- Includes a complete development environment

Linux Distributions

- There are over 100 different versions of Linux - called *distributions*



- Each *distribution* offers a unique combination of features and applications to suit the needs of different end users

Linux Distributions

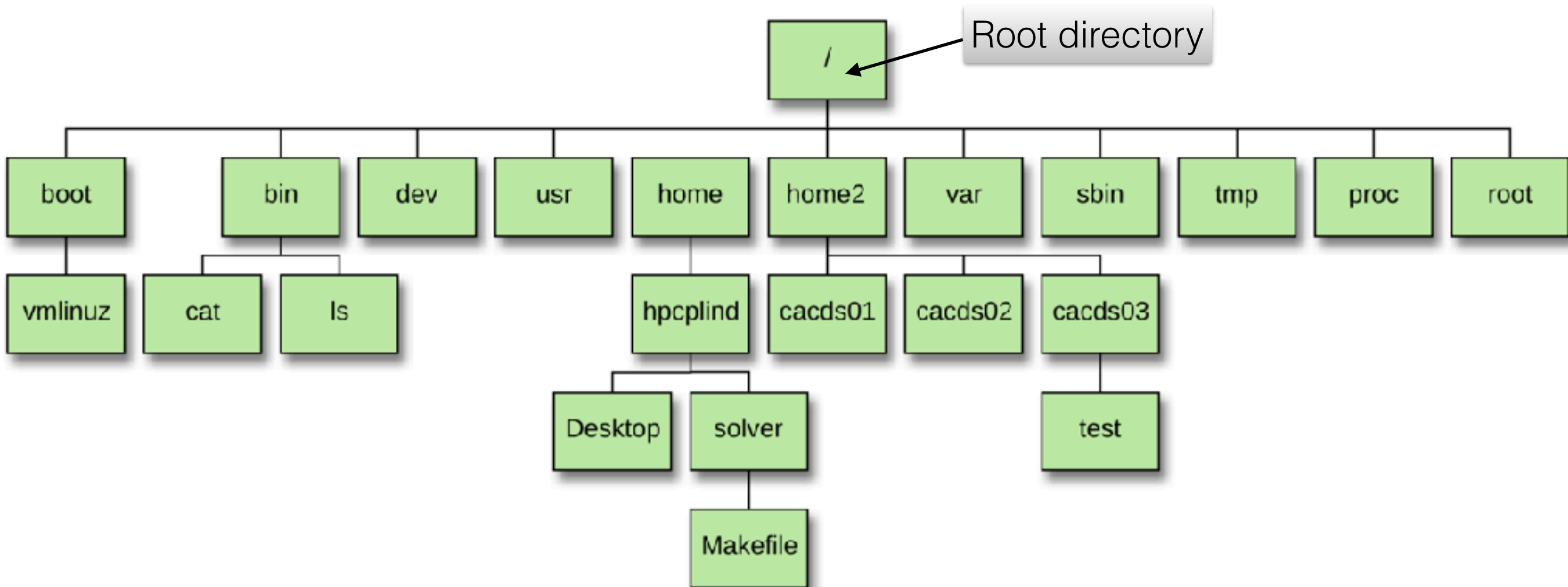
- <https://distrowatch.com/>
- provides news, comparisons, popularity ranking of various Linux distributions
- Helps to decide which distribution is a good fit

Page Hit Ranking		
Data span:		
Year 2016		Go
Rank	Distribution	HPD*
1	Mint	2905
2	Debian	1819
3	Ubuntu	1576
4	openSUSE	1277
5	Manjaro	1145
6	Fedora	1081
7	Zorin	944
8	elementary	927
9	CentOS	828
10	Arch	780
11	deepin	749
12	Mageia	682
13	PCLinuxOS	676
14	Ubuntu MATE	650
15	Android-x86	605
16	Antergos	587
17	Slackware	555
18	LXLE	549
19	Lite	542
20	Lubuntu	511

Linux File System

- A file system is the way files are organized on the disk
 - methods and data structures that an operating system uses to keep track of files on a disk or partition
- Linux uses several types of file systems
 - Open-source (completely customizable)
- Portable to nearly any hardware platform (cell phones, roku, steamOS devices, PS3, tablets, TVs, routers)
- Highly scalable to lots of cores, and or lots of memory
- Robust and proven security model
- Includes a complete development environment

File System Hierarchy



- Reminder: Full Path

Full PATH to "Desktop" folder in hpcplind's account

/home/hpcplind/Desktop

Full PATH to Makefile file

/home/hpcplind/solver/Makefile

Basic Unix commands (cont.)



- `change directory cd`
- `env` list all environment variables/settings
- `clear` clears printed content of terminal
- `who` print the list of all currently logged in users
- print summary of disk usage `df`

Arguments in commands



- `df -h` prints human readable disk usage
- `which df` shows the commands full path (helpful if different versions of a command are installed)
- `file file1` print the type of a file
- print summary of disk usage `df`

Data Transfer in Linux Systems

- Use *scp* (Secure Copy) for file and folder transfers to/from a remote server where you have a user account:

From *scp filename username@server:path_to_destination* *To*
scp username@server:path_to_remote_file path_to_destination_file

- Use *-r* option (recursive) if you want to transfer a directory

- Examples:

```
scp intro2linux.tar.gz cacds25@whale.cs.uh.edu:
```

```
scp Courses/Linux/Material/intro2linux.tar.gz cacds25@whale.cs.uh.edu:test
```

```
scp cacds25@whale.cs.uh.edu:intro2linux.tar.gz .
```

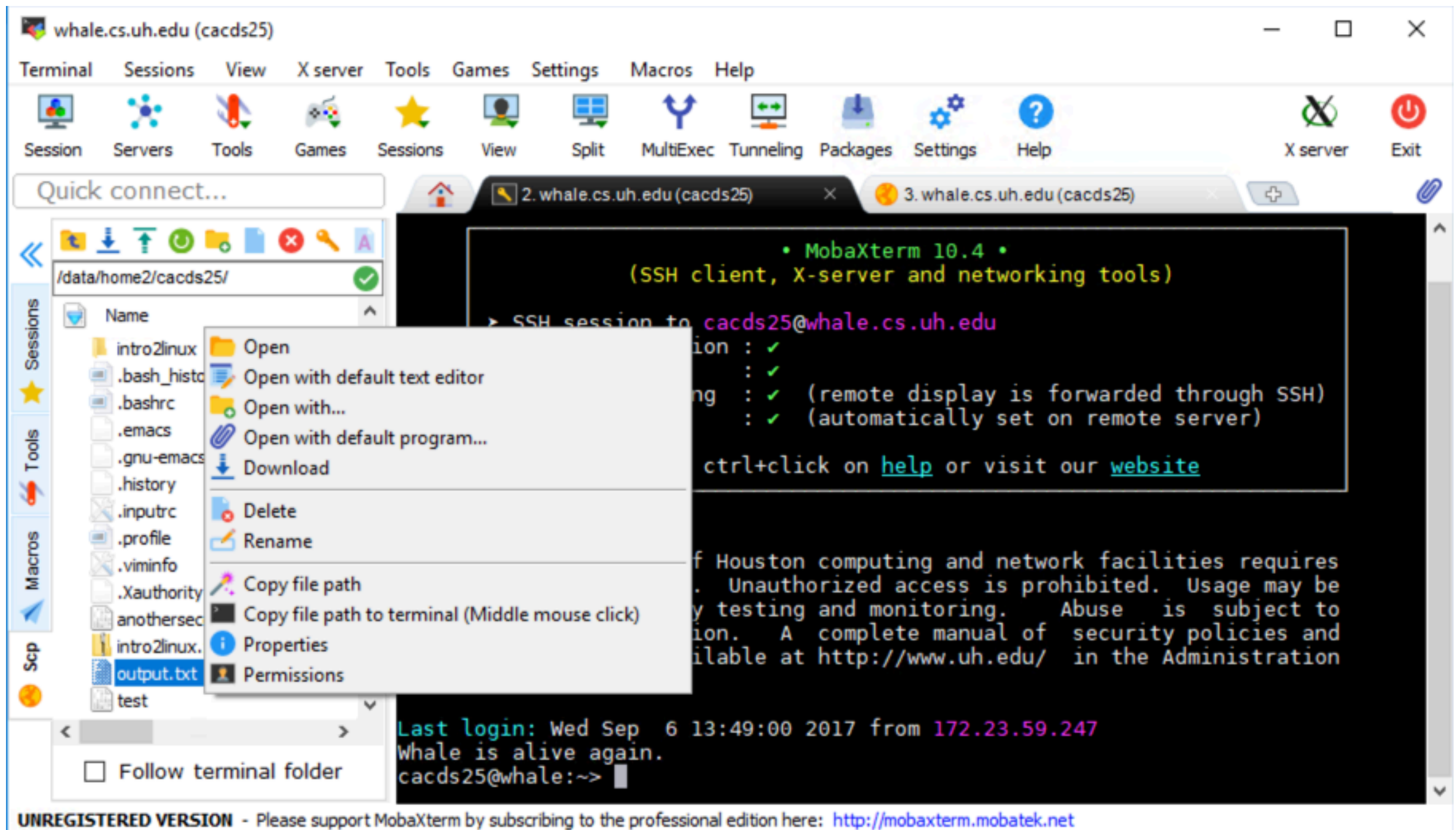
```
scp -r intro2linux cacds25@whale.cs.uh.edu:
```

```
scp -r cacds25@whale.cs.uh.edu:intro2linux .
```



- Task: Download the *intro2linux.tar.gz* files from the Moodle and transfer it into your account on *whale.cs.uh.edu*

Data Transfer from Windows



- Use MobaXterm and drag/drop

Archiving/Pack/Unpack

- Using the `tar` command
- unpacking/extracting an archive with option `-xvzf` == extract a compressed file archive in verbose mode, similar to `unzip`
- Example: `tar -xvzf intro2linux.tar.gz`
- Creating an archive with option `-cvzf` == create a compressed file archive in verbose mode
- Example: `tar -czvf intro2linux.tar.gz test_directory`
- Task: Extract the `intro2linux.tar.gz` file on `whale.cs.uh.edu`



Simple Data Processing

- Use commands like `head`, `tail`, `more`, `less` for first look at file
- Sorting can be done using `sort`
- Example:

```
> cd intro2linux  
> head z-a.txt  
> sort z-a.txt  
> sort -r a-z.txt
```



Combining streams

- `paste` command lets you merge two or more input streams side by side
- Example:
 - > `cat serial.txt`
 - > `cat data.txt`
 - > `paste serial.txt data.txt`



Cutting streams

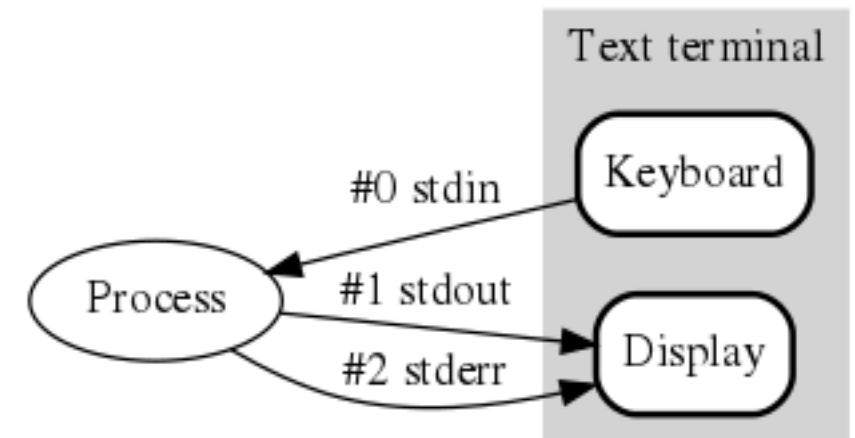
- Use the `cut` command to print out selected sections from each line of an input stream or file (needs to be tab separated if used without options)
- Use `cut` with the `-d` option to change the delimiter
- Example:

```
> cut -f 1,2 all.lanes.txt  
> cut -d ' ' -f 2- all.lanes2.txt
```



Further I/O redirection

- I/O redirection is a way of manipulating the input/output of Linux programs, allowing you to capture the output in a file, or send it to another program



- Use cut with the -d option to change the delimiter
- Example (store the first 9 lines into new file):

```
> head -n 9 dictionary.txt > temp.txt
```



Pipes

- Another useful technique is to redirect one program's output (stdout) into another program's input (stdin). This is done using a “pipe” character.
- Example:
 - > `cat z-a.txt | sort`
 - > `cat dictionary.txt`
 - > `cat dictionary.txt | grep ing`
 - > `cat dictionary.txt | grep ing | grep un`



Further usage of *grep*

- The `wget` command can download files directly from the web

```
> wget https://tinyurl.com/ybdmecbs -O intro2linux.zip
```

```
> unzip intro2linux.zip
```



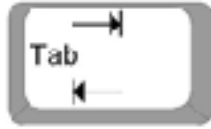
- Use `grep` to search a file for lines that do NOT contain pattern and prints result to stdout

```
> grep -v ing dictionary.txt
```

- Use `grep` to search using patterns stored in another file and prints them to stdout

```
> grep -f items2searchFor.txt dictionary.txt
```

Other useful basic Unix commands

- Don't forget to the TAB key  to autocomplete commands, paths, filenames etc.
- `top` will list processes/tasks running on your system (similar to task manager on windows)
- `q` or `CTRL-c` can help you get “unstuck”
- `tr` translate or delete characters

```
> echo linux | tr 'a-z' 'A-Z'
```

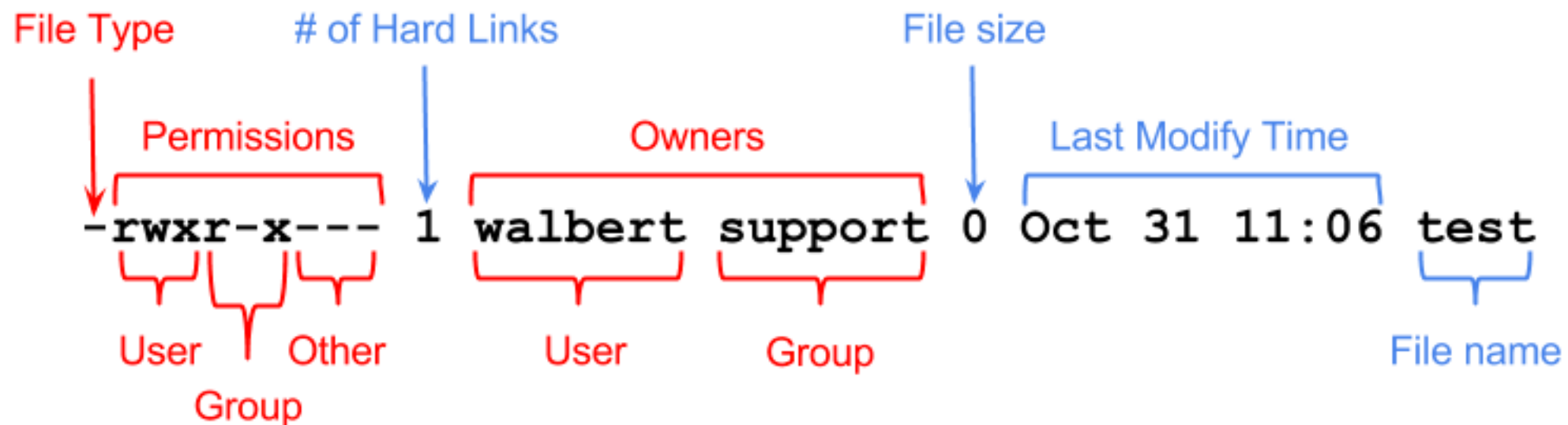
```
> echo 'world}}}' | tr '}' '!'
```

```
> echo 'world}}}' | tr -d '}'
```



File/Directory Permissions

- Three user types are associated with file permissions: Owner(u), Group (g) and Other/world (o)
- They can be seen when we use `ls -l`



File/Directory Permissions

- Control access to files & directories by setting permissions
- Setting permissions using read /write or executable options with the `chmod` command

`chmod ug+r file` makes a file readable by owner (u) and group (g)

`chmod ug+w file` writes to the file are permitted

`chmod ug+x file` makes a file executable

`chmod ug+rx file` makes a file executable, writable and readable

`chmod ugo+r file` makes a file readable by owner (u) and group (g) and world(o)

- For directories you apply the recursive option `-R`

`chmod -R u+rx directory` makes a directory readable

Change File Permissions- Octal Notation

- File permissions can be changed using octal notation

- “-rwxr-xr-x” = 755
- “-rw-rw-r--” = 664
- “-r-x-----” = 500

- Example:

```
> chmod 755 dictionary.txt
```

```
> chmod -R 755 ../intro2linux/
```

	4	2	1	
0	-	-	-	no permissions
1	-	-	x	only execute
2	-	w	-	only write
3	-	w	x	write and execute
4	r	-	-	only read
5	r	-	x	read and execute
6	r	w	-	read and write
7	r	w	x	read, write and execute

Command Line Expansion

- Bash can transform the command-line input using expansions

- **Brace expansion** {begin..end} or {begin..end..increment}

```
> echo {Z..A}
```

```
> echo Front-{A,B,C}-Back
```

- **Arithmetic expansion** \$((expression))

```
> AA=50
```

```
> echo $ ( (AA++) )
```

```
> BB=$(( AA*2 ))
```

```
> echo $BB
```

- **Parameter expansion** (mostly useful in scripts)

```
> echo $USER
```

```
> echo $HOME
```



Example Brace Extension

```
> mkdir Photos  
> cd Photos  
> mkdir {2011..2013}-{01..12}  
> ls
```



```
> 2011-01    2011-05    2011-09    2012-01  
2012-05    2012-09    2013-01    2013-05    2013-09  
2011-02    2011-06    2011-10    2012-02    2012-06  
2012-10    2013-02    2013-06    2013-10  
2011-03    2011-07    2011-11    2012-03    2012-07  
2012-11    2013-03    2013-07    2013-11  
2011-04    2011-08    2011-12    2012-04    2012-08  
2012-12    2013-04    2013-08    2013-12
```

Exercise



- Write a 1 line command to download 13 human genomic DNA digest files (format md5) from the public medical (pubmed) FTP site datasets to download range from
human_genomic00.tar.gz.md5 - human_genomic.12.tar.gz.md5
- ftp site/URL folder = `ftp://ftp.ncbi.nlm.nih.gov/blast/db/`
- Hint: use a combination of the `wget` command and brace expansion

Exercise



- Write a 1 line command to download 13 human genomic DNA digest files (format md5) from the public medical (pubmed) FTP site datasets to download range from
human_genomic00.tar.gz.md5 - human_genomic.12.tar.gz.md5
- ftp site/URL folder = `ftp://ftp.ncbi.nlm.nih.gov/blast/db/`
- Hint: use a combination of the `wget` command and brace expansion
- `wget ftp://ftp.ncbi.nlm.nih.gov/blast/db/human_genomic.{00..12}.tar.gz.md5`

Command Substitution

- The command inside the is run, and anything the command writes to standard output is returned as the value of the expression. These constructs can be nested, i.e., the UNIX command can contain command substitutions

- Syntax A) *\$(Linux_command)*

```
> export DATE=$(date)
> echo "Right now it's $DATE"
```

```
> Right now it's Thu May 17 23:13:52 EDT 2012
```

- Syntax B) *`Unix Command`*

```
> export DATE=`date`
> echo "Right now it's $DATE"
```

```
> Right now it's Thu May 17 23:13:52 EDT 2012
```



Arrays & Random numbers



- Arrays are variables that hold more than one value at a time.
- Arrays in bash are limited to a single dimension.

```
> POWERBALL=(54 69 23 66 78 99)
> echo "one of my lucky number is " ${POWERBALL[3]}
> one of my lucky number is 66
```

- Make use of the RANDOM bash variable

```
> echo $(( RANDOM%= 7 ))
> 6
> echo "one of my lucky number is " ${POWERBALL[$(( RANDOM%=
7 ))]}
```

- Access arrays using indexing

```
> foo=({20..35})
> echo ${foo[2]}
> echo ${foo[@]}
```

Shell Scripts.

Shell Scripts

- Shell script is a file containing a series of commands
- shell reads this file and carries out the commands as if they were entered on the shell prompt
- Steps for creating a shell script

1. Write the shell script.

- Shell scripts are ordinary text files
- Use a text editor to write them

2. Add executable permission to the script file
(e.g., `script.txt`)

```
#!/bin/sh
# this is a comment
body of program
to continue a line append \, file
this is the rest of the continuation
exit 0
```

Shell Script Example

- Use vi to create the following script and store it as `script.sh`

```
#!/bin/bash
cd $HOME
tar -xf intro2linux.tgz
tar -cvzf example.tar.gz intro2linux
mkdir dustbin
mv example.tar.gz ./dustbin
cd dustbin
tar -xvf example.tar.gz ; mv
intro2linux newdir ls newdir >
contents.txt
cd $HOME
```



- then make the file executable changing the permissions.
`chmod u+x script.sh`
- execute the file `./script.sh`

Functions

- Perform same task as shell scripts but are much faster as shell saves them in memory for use

- Syntax:

```
[function] function-name ()  
{  
    commands  
}
```

- word “function” is optional
- function-name is the name you use to call the function
- commands comprise the list of commands the function executes when you call it

Function Example



```
function myscript () {  
    cd $HOME  
    tar -xf intro2linux.tgz  
    tar -cvzf example.tar.gz intro2linux  
    mkdir dustbin  
    mv example.tar.gz ./dustbin  
    cd dustbin  
    tar -xvf example.tar.gz ; mv intro2linux    newdir  
    ls newdir > contents.txt  
    cd $HOME  
}
```

Calling Functions

```
#!/bin/bash

function myscript () {
    cd $HOME
    . . . . .
}

myscript
```

Parameters



- Make your scripts more functional

```
#!/bin/sh
# pars
echo "There are $# parameters." echo "The parameters are
$@"
echo "The script name is $0"
echo "The first parameter is $1" echo "The second
parameter is $2" exit 0
```

```
$ pars apple orange
```

```
There are 2 parameters.
The parameters are apple orange The script name is ./
pars
The first parameter is apple The second parameter is
orange
```

```
$
```

Flow control

- add intelligence to our scripts
- Programs need to make decisions and perform different actions depending on conditions
- The shell provides several commands that we can use to control the flow of execution in our program.
- If...else, for loops,

Flow Control - if...

- The if command is fairly simple on the surface; it makes a decision based on the exit status of a command.
- Syntax:
if commands; then
commands
[elif commands; then
commands...]
[else
commands]
fi (where commands is a list of commands)
- *Exit status:*
 - Commands (including the scripts and shell functions we write) issue a value to the system when they terminate, called an exit status.
 - This value, which is an integer in the range of 0 to 255, indicates the success or failure of the command's execution.
 - By convention, a value of zero indicates success and any other value indicates failure. The shell provides a parameter that we can use to examine the exit status.

Exit Status

```
> [me@linuxbox ~]$ ls -d /usr/bin
```

```
> /usr/bin
```

```
> [me@linuxbox ~]$ echo $?
```

```
> 0
```

```
> [me@linuxbox ~]$ ls -d /bin/usr
```

```
> ls: cannot access /bin/usr: No such file or  
directory
```

```
> [me@linuxbox ~]$ echo $?
```

```
> 2
```

Exit Status Example 1

```
#!/bin/sh
# rem
rm junk
echo "The return code from rm was $?"
exit 0
```

```
$ touch junk
$ rem
```

The return code from rm was 0

```
$ rem
```

```
rm: junk: No such file or directory
The return code from rm was 2
```



Exit Status Example 2

```
#!/bin/sh
# quiet
rm junk 2> /dev/null
echo "The return code from rm was $?"
exit 0
```

```
$ touch junk
$ quiet
The return code from rm was 0
```

```
$ quiet
The return code from rm was 2
```



true/false

- The shell provides two extremely simple builtin commands that do nothing except terminate with either a zero or one exit status.
- The `true` command always executes successfully and the `false` command always executes unsuccessfully:

```
> [me@linuxbox~]$ true
```

```
> [me@linuxbox~]$ echo $?
```

```
> 0
```

```
> [me@linuxbox~]$ false
```

```
> [me@linuxbox~]$ echo $?
```

```
> 1
```

true/false

- We can use these commands to see how the if statement works. What the if statement really does is evaluate the success or failure of commands:

```
> [me@linuxbox ~]$ if true; then echo  
  "It's true."; fi
```

```
> It's true.
```

```
> [me@linuxbox ~]$ if false; then echo  
  "It's true."; fi
```

```
> [me@linuxbox ~]$
```

test

- The *test* command is used most often with the *if* command to perform true/false decisions.
- The command is unusual in that it has two different syntactic forms:

First form

`test expression`

Second form

`[expression]`

- The test command works simply. If the given expression is true, test exits with a status of zero; otherwise it exits with a status of 1.

```
if [ -f .bash_profile ]; then
```

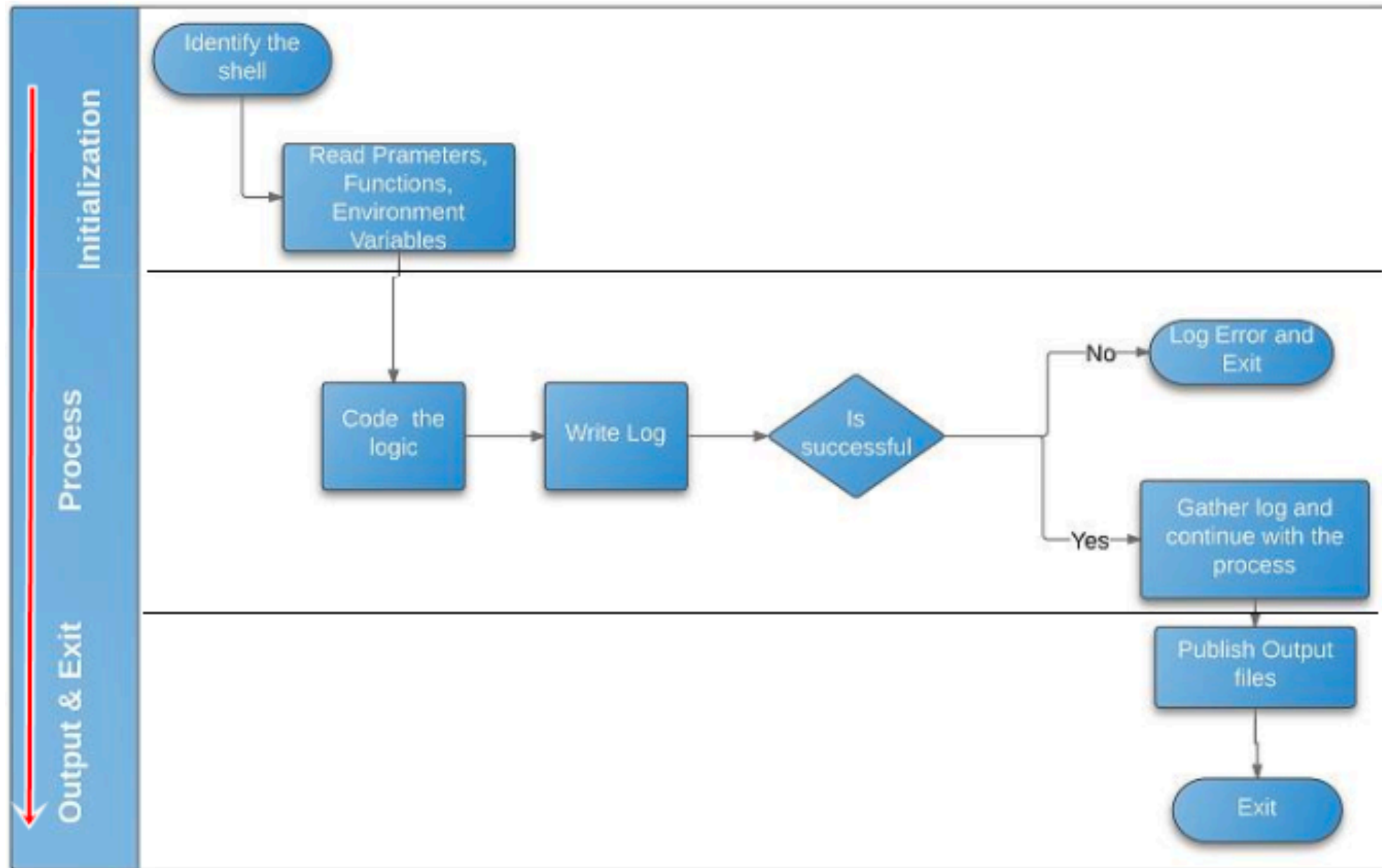
```
    echo "You have a .bash_profile. Things are fine."
```

```
else
```

```
    echo "Yikes! You have no .bash_profile!"
```

```
fi
```

Shell Scripting Programming Flow



Flow Control - For loops

- Syntax:

for *loop-index* in [argument
list] **do** *command(s)...*
done

- Example (*loop1.sh*):

```
for i in $(seq 1 10)
do
    echo -n This is iteration $i
    echo -n " and the time is "
        date +%T
        sleep 1
done
```

- Example (*loop1.bash*):

```
for i in $(seq 1 10)
do
    echo -n This is iteration $i
    echo -n " and the time is "
        date +%T
        sleep 1
done
```



Flow Control - For loops

- Another way in C/C++ Style:
- Example (*loop2.sh*):

```
#!/bin/bash
```

```
for ((i=1; i<=10; i++))
```

```
do
```

```
    echo -n This is iteration "$i"
```

```
    echo -n " and the time is "
```

```
date +%T done
```



Common Problems in for loops

```
for i in {1..10}

    echo -n This is iteration $i
    echo -n " and the time is "
    date +%T
done
```

Missing “do” command

```
for ((i=1; i<=10; i++))
do
    echo -n This is iteration $i
    echo -n " and the time is "
    date +%T
```

Missing “done” command

For loop Conditional “if” and “test”

```
for i in {1..10}
do
    echo -n This is iteration $i
    if [ $i -eq 5 ]
    then break
fi
done
```

```
for ((i=1; i<=10; i++))
do
    echo -n This is iteration $i
    test $i -eq 5 && break
done
```



- See *flow-control-loop1.sh* *flow-control-loop2.sh* & *flow-control-loop3.sh*

Accessing arrays with for loop @

- Example:

```
animals=("a cat" "ate my" "yellow fish")
```

```
for i in "${animals[@]}"
```

```
do
```

```
    echo $i
```

```
done
```

```
for ( (i=0; i<=2; i++) )
```

```
do
```

```
    echo ${animals[$i]}
```

```
done
```



Some Gotchas

- Never use `test` as the name of a variable or a shell script file.
- When using `=` as an assignment operator, do not put blanks around it.
- When using `=` as a comparison operator, you must put blanks around it.
- When using `if []` put spaces around the brackets (except after `]` when it is the last character on the line).

Final Example

```
#!/bin/sh
# list names of all files containing given words
if [ $# -eq 0 ]
then
    echo "findtext word1 word2 word3 ..."
    echo "lists names of files containing all given words"
    exit 1
fi
for fyle in *
do
    bad=0
    for word in $*
    do
        grep $word $fyle > /dev/null 2> /dev/null
        if [ $? -ne 0 ]
        then
            bad=1
        fi
    done
    if [ $bad -eq 0 ]
    then
        echo $fyle fi
    done
done
exit 0
```

More Unix Commands (useful in shell scripts)

- `basename` extract file name from path name
- `cmp -s` to compare files (silently)
- `expr` (evaluate an expression)
- `mail` send email (if enabled on server)
- `sleep` to suspend execution for given time
- `find` search files and directories
- `shift -n` Shift positional parameters to the left by `n`

Further Reading

- *The Linux Command Line: A Complete Introduction*
Paperback by Shotts
- *Practical Guide to Linux Commands, Editors, and Shell Programming* by Sobell
- *Learning the bash Shell: Unix Shell Programming (In a Nutshell* (O'Reilly))
- Free Ebooks
Advanced Bash-Scripting Guide
<http://tldp.org/LDP/abs/html/>

Bash Guide for Beginners

<http://tldp.org/LDP/Bash-Beginners-Guide/html/>