CIS 191

Linux and Unix Skills

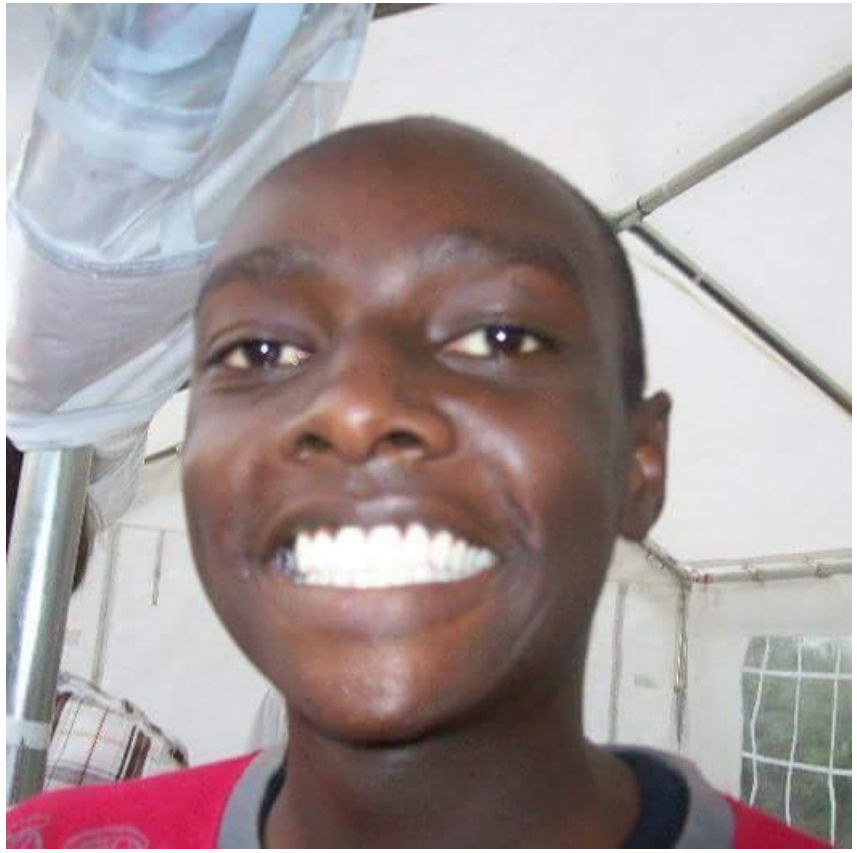
LECTURE 1

Lecture Plan

1. Introduction and Logistics.
2. Unix/Linux Overview.
3. Navigating the Filesystem.
4. Using the Terminal Efficiently.

Introduction and Logistics

About Me



Instructor: Solomon Maina
Email: smaina@seas.upenn.edu
Office Hours : By appointment
Location : Levine 057

Why Take CIS 191?

1. Learn how to use the Unix OS using the command line.
2. Learn how to write shell and Python scripts.
3. Learn some important tools and skills for programming.
4. Get a glimpse of how Unix/Linux works.

What is a Script?

- A ***script*** is a file containing code that is executed by a program known as an ***interpreter***, without being compiled into an executable file executed by the CPU.
- Examples are shell scripts interpreted by command-line interpreters like bash, Python scripts interpreted by the Python interpreter, JavaScripts interpreted by JavaScript engines etc.

Shell Script Example

The script below renames filenames of the form

24-09-2007-picturename.jpg

to

2007-09-24-picturename.jpg

```
for fn in *.jpg
do
        mv "$fn" "$(echo "$fn" |\
        sed –r 's/([0-9]+)-([0-9]+)-([0-9]+)/\3-\2-\1/')"
done
```

Workload

- Lectures once a week.
- Four assignments, roughly one every two weeks.
- Final project.

Resources

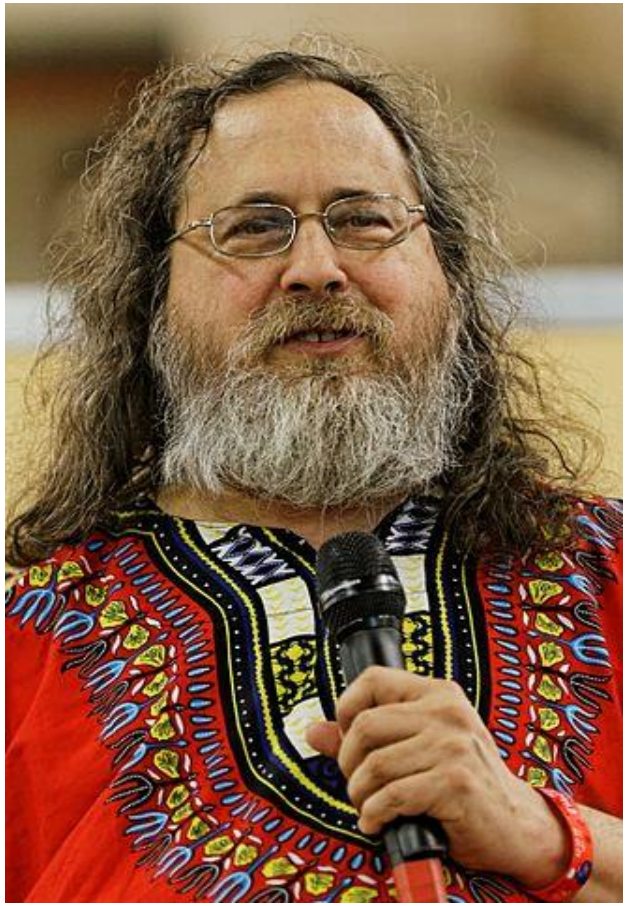
- Course website: https://www.cis.upenn.edu/~cis191/
- Piazza: https://www.piazza.com/upenn/fall2019/cis191
- Canvas: https://canvas.upenn.edu/

Unix/Linux Overview

What is Unix?

- Unix is a widely influential operating system developed in the 70s.
- Unix was written in C instead of assembly, making it the first portable OS.
- Unix helped set the standard for multi-tasking, multi-user systems.
- Unix was proprietary!

GNU: GNU's Not Unix!



- Richard Stallman started the GNU project in 1983.
- Goal: create a ***free*** complete Unix-compatible software system; free as in the users have the freedom to run, copy, distribute, study, change and improve the software.
- Largely achieved this goal by the early 1990s except for some low-level elements such as the kernel.

Linux

- Linus Torvalds wrote a kernel (with others) in 1991.
- Kernel adopted by the GNU system in 1992.
- GNU/Linux system now just called "Linux".

Linux Distros

- Linux comes in hundreds of flavors called "distributions" or ***distros***.
- Distros generally have different design goals (security, speed, desktop use, etc).
- Examples are Ubuntu, Mint, Debian, Fedora, Red Hat, openSUSE etc…

Unix Flavours

- There currently exist many operating systems based on Unix, or ***Unix flavours*** e.g.
  - Berkeley Software Distribution (BSD)
  - GNU/Linux
  - Mac OS X
  - Sun's Solaris
  - IBM AIX, HP-UX, Silicon Graphics IRIX, etc…
- For CIS 191, we will focus on GNU/Linux.

SUS and POSIX

- The ***Single UNIX Specification*** (SUS) is the collective name of a family of standards for computer operating systems.
- Operating systems that conform to SUS gain the "UNIX" trademark.
- In 1988, these standards became ***POSIX***, which loosely stands for Portable Operating System Interface.
- Some Unix flavours are POSIX-certified (e.g. Mac OS X) while others are not (e.g. Linux).

Shells

- Unix flavours allow the user to interact with the OS through command line interpreters, or ***shells***.
- There are many shells:
  - sh – Bourne shell (Unix)
  - bash – Bourne Again shell (GNU/Linux)
  - csh – C shell (BSD Unix)
  - ash, dash, ksh, zsh, tcsh, fish etc…
- For CIS 191, we will focus on bash.

Getting Unix/Linux

- Dual boot i.e. run your OS and Unix/Linux side by side but not at the same time.
- Install a virtual machine (recommended).
- Install Windows Subsystem for Linux (WSL) on Windows.
- Download Cygwin on Windows.
- For CIS 191, you ***MAY*** get by using Mac OS X on Mac, Cygwin on Windows, or any other Unix flavor.

Standard Argument Syntax

- The standard argument syntax for commands is as follows:

  command [-a][-b][-c option_argument] [-d|-e][-f[option_argument]][operand...]

- command is the name of the command.
- The arguments that consist of '-' characters and single letters or digits, such as 'a', are known as ***options*** or ***flags***.
- Certain options are followed by an ***option argument***, as shown with [-c option_argument].
- Arguments or option-arguments enclosed in the '[' and ']' notation are optional and can be omitted.
- Arguments separated by the '|' bar notation are mutually-exclusive.
- Ellipses ("...") are used to denote that one or more occurrences of an operand are allowed.
- The order of different options relative to one another often does not matter.
- Options are usually required to precede operands on the command line.

Common Options

- **`-h, --help`** : Give a usage message and exit
- **`-v, --version`** : Show program version and exit
- **`-a, --all`** : Show all information or operate on all arguments
- **`-l, --list`** : List files or arguments without taking other action
- **`-o`** : Output filename
- **`-r, -R, --recursive`** : Operate recursively (down directory tree)
- **`-q, --quiet`** : Suppress stdout
- **`-v, --verbose`** : Output additional information to stdout or stderr
- **`-z, --compress`** : Apply compression
- Most commands allow you to give one or more options without option-arguments, followed by at most one option that takes an option-argument group behind one '-' delimiter. e.g.

```
ld –do output_file = ld –d –o output_file
```

Navigating the Filesystem

Navigating the Filesystem

`ls [OPTION]… [FILE]…`

- List information about the FILEs (the current directory by default)
- **-a** : do not ignore entries starting with a . (hidden files).
- **-l** : use a long listing format.

Navigating Directories

pwd [OPTION]…

- Print name of current/working directory.

cd [OPTION]… [DIRECTORY]

- Change the working directory to DIRECTORY.

mkdir [OPTION]… [DIRECTORY]…

- Make the DIRECTORY(ies), if they do not already exist.

rmdir [OPTION]… DIRECTORY…

- Remove the DIRECTORY(ies), if they are empty.

The Directory Stack

- The directory stack is a list of recently-visited directories.
- The ***pushd*** command adds directories to the stack as it changes the current directory.
- The ***popd*** command removes specified directories from the stack and changes the current directory to the directory removed.
- The current directory is always at the "top" of the stack.
- The ***dirs*** command displays the contents of the directory stack.
- Use the directory stack when navigating between parts of the file-system that are far from each other

The Directory Stack

dirs [-clpv] [+N | -N]

- Display the list of currently remembered directories.
- **-c** : Clears the directory stack by deleting all of the elements.
- **-l** : Produces a listing using full pathnames; the default listing format uses a tilde to denote the home directory.
- **-p** : Causes **dirs** to print the directory stack with one entry per line.
- **-v** : Causes **dirs** to print the directory stack with one entry per line, prefixing each entry with its index in the stack.
- **+N** : Displays the Nth directory from the left.
- **-N** : Displays the Nth directory from the right.

., .., ~

- On Unix-like operating systems, every directory contains an object represented by a single dot and another represented by two successive dots.
- The former refers to the directory itself and the latter refers to its *parent directory* (i.e., the directory that contains it).
- In most shells, tilde (~) expands to the value of the home directory of the current user.
- ~user expands to the value of the home directory of the user named user.
- ~+N expands to the string that would be displayed by `'dirs +N'`.
- ~-N expands to the string that would be displayed by `'dirs -N'`.

Creating and Viewing Files

touch [OPTION]… FILE…

- Update the access and modification times of each FILE to the current time.
- A FILE argument that does not exist is created.

cat [OPTION]… FILE

- Concatenate FILE(s) to standard output
- With no FILE, or when FILE is -, read standard input.

Viewing Files

`more [OPTION]… FILE…`

- Allows you to page through text one screenful at a time.
- SPACE, z or RETURN to display next page, d to scroll forward.
- b to display previous page, but no way to scroll backward.
- q to quit

`less [OPTION]… FILE`

- Same as **`more`** but lets you scroll backwards.
- Does not have to read the entire input file before starting, hence is faster with large files.

Viewing Files

head [OPTION]… FILE…

- Print the first 10 lines of FILE to standard output.
- **-n** NUM: Print the first NUM lines instead of the first 10; with the leading '-', print all but the last NUM lines of each file.
- With no FILE, or when file is -, read standard input.

tail [OPTION]… FILE

- Print the last 10 lines of FILE to standard output.

Deleting Files

rm [OPTION]… [FILE]…

- Remove FILE.
- **-r, -R, --recursive** : removes directories and their contents recursively.
- **-i** : prompts the user whether to proceed with the entire operation if there are more than three files or **-r, -R, --recursive** are given.

Copying and Moving Files

cp [OPTION]… SOURCE DEST

- Copy SOURCE to DEST.

mv [OPTION]… SOURCE DEST

mv [OPTION]… SOURCE DIRECTORY

- Rename SOURCE to DEST, or move SOURCE(s) to DIRECTORY.

Using the Terminal Efficiently

clear, exit, man, completion

- **clear** (or CTRL + L) clears the terminal screen.
  - This can also be achieved by CTRL + L.
- **exit** (or CTRL – D) terminates the terminal session.
  - This can also be achieved by CTRL + D.
- **man** command displays the reference manual page for command.
- If you press TAB while typing, Bash attempts ***completion***, treating the text as a variable (if the text begins with '$'), username (if the text begins with '~'), hostname (if the text begins with '@'), or command (including aliases and functions) in turn, and if none of these produces a match, filename completion is attempted.

Command History and History Expansion

- The ***command history*** is the list of commands previously typed in the shell.
- ***History expansions*** introduce words from the command history into the input stream.
- History expansions are introduced by the appearance of the history expansion character, which is '!' by default e.g.
  - !! - Refer to the previous command.
  - !-n - Refer to the command n lines back.
  - !-string - Refer to the most recent command preceding the current position in the history list starting with string.
- Type CTRL + P or up arrow to view the previous command.
- Type CTRL + N or down arrow to access the next command.

Searching History

- Type CTRL + R followed by a search string; matching history entries will be shown.
- Type CTRL + R again to see other matches.
- If you like an entry, type ENTER to execute it.
- Type ESC to copy the entry to the prompt without executing.
- Type CTRL + G to exit search and go back to an empty prompt.

Aliasing

alias [-p]… [name[=value] …]

- Defines or display aliases (an alias is an alternative name for a command).
- **alias** prints the list of aliases in the reusable form 'alias NAME=VALUE' on standard output if no arguments are given, otherwise an alias is defined for each NAME whose VALUE is given e.g. typing

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

will cause the name `ll` to behave like the command `ls -l`

unalias [-a] [name …]

- Remove each name from the list of aliases.
- If **-a** is supplied, all aliases are removed.

Configuring Bash

- When an interactive shell that is not a login shell is started, Bash reads and executes commands from ~/.bashrc, if that file exists.
- Edit the .bashrc file in your home directory to configure bash.
- Alternatively, you may edit one of these other configuration files:
  - /etc/profile : a global configuration file executed when bash is invoked as an interactive login shell, or as a non-interactive shell with the **--login** option.
  - ~/.bash_profile, ~/.bash_login, ~/.profile : per-user files executed when bash is invoked as an interactive login shell, or as a non-interactive shell with the **--login** option; only the first one that exists in that order and is readable is executed.
  - ~/.bash_logout : executed when an interactive login shell exits, or a non-interactive login shell with the **--login** option executes the **exit** command.

Configuring Bash

- Some common ways of configuring bash are:
  - Setting aliases e.g. setting `ll` to be an alias for `ls -l`,
  - Customizing the PS1 variable to change the prompt at the beginning of the line, each time you hit enter on the command line,
  - Customizing the PROMPT_COMMAND variable; this command is executed before the printing of PS1,
  - Changing the colour of PS1 or PROMPT_COMMAND
  - Adding programs to the PATH variable, which contains the directories searched when you execute a command,
  - Setting the EDITOR variable to a preferred text editor e.g. vim or emacs,
  - etc!

Cursor Movement Keyboard Shortcuts

Shortcut	Description
CTRL + A or Home	Go to the beginning of the line.
CTRL + E or End	Go to the end of the line.
ALT + B	Move cursor back one word.
ALT + F	Move cursor forward one word.
CTRL + F or Right Arrow	Move cursor forward one character.
CTRL + B or Left Arrow	Move cursor backward one character.

Editing Keyboard Shortcuts

Shortcut	Description
CTRL + L	Clear the Screen, similar to the clear command
ALT + Backspace	Delete the word before the cursor.
ALT + D	Delete the word after the cursor.
CTRL + D	Delete the character after the cursor.
CTRL + H or Backspace	Delete the character before the cursor.
CTRL + W	Cut the word before the cursor.
CTRL + K	Cut the line after the cursor.
CTRL + U	Cut the line before the cursor.
CTRL + Y	Yank (paste) the last thing to be cut.
ALT + Y	Yank-pop i.e. cycle through clipboard (only applies if last command was a yank or yank pop)
CTRL + _ or CTRL + X CTRL + U	Undo.
TAB	Tab completion for file/directory names.

CIS 191

Linux and Unix Skills

LECTURE 2

Lecture Plan

1. Working with Files.
2. Manipulating Text.

Working with Files

Unix Files

- The Unix philosophy is that ***everything is a file***, in that many components of the operating system have a file representation in the filesystem, including:
  - Block devices such as hard disks and optical drives,
  - Character devices such keyboards and terminal devices,
  - Directories,
  - etc.
- ***file*** <filename> shows the type of filename.
- The Unix file-system is hierarchical tree structure: directories can contain directories which can contain directories etc, and every file has only one parent.
- Most Linux distributions follow the ***Filesystem Hierarchy Standard (FHS)***, which consists of a set of requirements and guidelines for file and directory placement under UNIX-like operating systems

FHS Required Directories	
Directory	***Description***
bin	Essential user command binaries for use by all users e.g. cat, cp, ls
boot	Static files of the boot loader
dev	Device files
etc	Host-specific system configuration
lib	Essential shared libraries and kernel modules
media	Mount point for removable media
mnt	Mount point for mounting a filesystem temporarily
opt	Add-on application software packages
run	Data relevant to running processes
sbin	Essential system binaries
srv	Data for services provided by this system
tmp	Temporary files
usr	Secondary hierarchy
var	Variable data

Mounting and Unmounting

mount -t type device dir

- The **mount** command attaches the filesystem found on device (which is of type type) at the directory dir.
- Conversely, the **umount** command will detach it again.

umount [OPTION] {directory|device}...

- The **umount** command detaches the mentioned file system(s) from the file hierarchy.
- A file system is specified by giving the directory where it has been mounted.

df [OPTION]… [FILE]…

- **df** displays the amount of disk space available on the file system containing each file name argument.
- If no file name is given, the space available on all currently mounted file systems is shown.

Users and Groups

- Unix-like operating systems are multi-user.
- Users may belong to a ***group***.
- Each file is owned by a unique user and group.
- Use of a files by a user is restricted in different ways depending on the owning user and group of the file/folder.
- Many operating systems have a special user conventionally called ***root*** who has all rights or permissions in all modes.

Users and Groups

- It is often recommended that no-one use root as their normal user account to prevent major damage to the OS.
- The command ***su*** [user], short for substitute user, allows a one to interact with the shell as user.
- The command ***sudo*** allows a user to run a command with the security privileges of another user, by default root.
- Some operating systems automatically give the initial user created the ability to run as root via sudo.

Permissions

- ***Modes*** are the file system permissions give to "user", "group" and "others" classes to access files under Unix.
- Permissions for regular files are represented symbolically in the form –rwxrwxrwx.
  - User's permissions are the first three characters.
  - Group's permissions are the middle three characters.
  - Other's permissions are the last three characters.
- 'r' means read permissions, 'w' write permissions and 'x' execute permissions.
- Permissions for files of non-regular type begin with a character different from '–' e.g. directory permissions begin with a 'd' instead of a '–'.
- -rw-rw-r-- means user and group can read and write the file while everyone else can only read it.

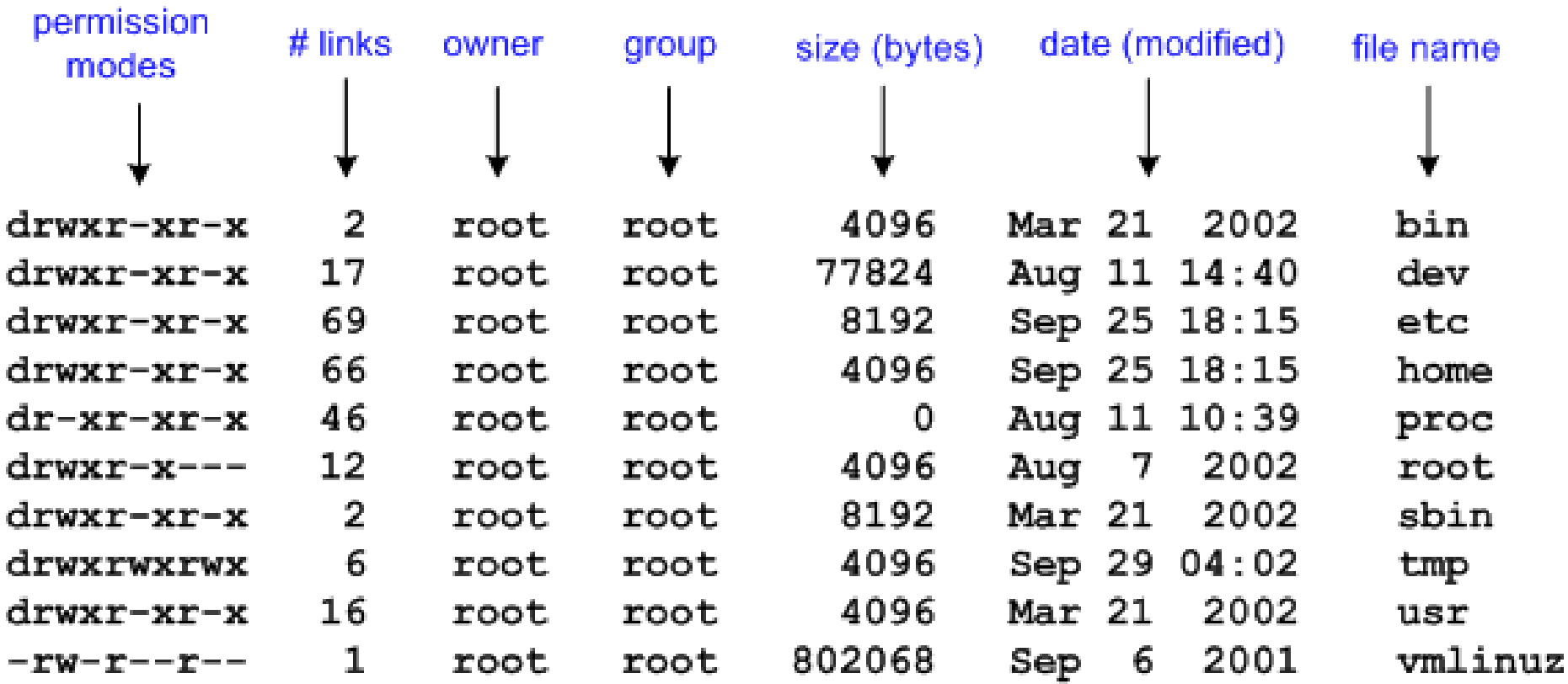
Permissions

- Permissions can also be represented by a 3 digit octal (base-8) number.

Octal Digit	Symbolic Notation	English
0	---	No permissions
1	--x	Execute permissions only
2	-w-	Write permissions only
3	-wx	Write and execute permissions only
4	r--	Read permissions only
5	r-x	Read and execute permissions only
6	rw-	Read and write permissions only
7	rwx	Read, write and execute permissions

- 755 means user has read, write and execute permissions, group has read and execute permissions only, and others have read and execute permissions only.

Discovering Permissions




```
drwxr-xr-x   2   root   root     4096   Mar 21  2002   bin
drwxr-xr-x  17   root   root    77824   Aug 11 14:40   dev
drwxr-xr-x  69   root   root     8192   Sep 25 18:15   etc
drwxr-xr-x  66   root   root     4096   Sep 25 18:15   home
dr-xr-xr-x  46   root   root        0   Aug 11 10:39   proc
drwxr-x---  12   root   root     4096   Aug  7  2002   root
drwxr-xr-x   2   root   root     8192   Mar 21  2002   sbin
drwxrwxrwx   6   root   root     4096   Sep 29 04:02   tmp
drwxr-xr-x  16   root   root     4096   Mar 21  2002   usr
-rw-r--r--   1   root   root   802068   Sep  6  2001   vmlinuz
```

Changing Permissions

chmod [OPTION]… MODE… FILE…

chmod [OPTION]… OCTAL-MODE… FILE…

- **chmod** changes the file mode bits of each given file according to MODE, which can be symbolic or numeric.
- A combination of the letters **ugoa** controls which users' access to the file will be changed: the user who owns it (**u**), other users in the file's group (**g**), other users not in the file's group (**o**), or all users (**a**). If none of these are given, the effect is as if (**a**) were given.
- The operator **+** causes the selected file mode bits to be added to the existing file mode bits of each file; **-** causes them to be removed; and **=** causes them to be added and causes unmentioned bits to be removed.

chmod Examples

- `chmod ug+rx file:` adds read and execute permissions for user and group.
- `chmod a-r file:` removes read access for everyone.
- `chmod ugo-rwx file:` removes all permissions.
- `chmod o=r file:` gives others only read permissions.
- `chmod 755 file:` changes the mode to `rwxr-xr-x`
- `chmod 600 file:` changes the mode to rw-------

Adding and Removing Users and Groups

adduser, addgroup, deluser, delgroup

- **adduser** and **addgroup** add users and groups to the system respectievely while **deluser** and **delgroup** deleter users and groups from the system e.g. the command

```
adduser blah
```

adds a user with the username "blah" while the command

```
addgroup blahs
```

adds a user group with the groupname "blahs".

- If called with two non-option arguments, **adduser** and **deluser** respectively add and remove an existing user to/from an existing group e.g. the command

```
deluser blah blahs
```

deletes the user named "blah" to the group named "blahs".

Changing Groups and Owners

chgrp [OPTION]… GROUP FILE

- Changes the group of each FILE to GROUP e.g. the following command changes the group of the file named "foo" to the group named "winners":

```
chgrp winners foo
```

chown [OPTION]… [OWNER] [:GROUP] FILE

- **chown** changes the user and/or group ownership of each given file e.g. the following command changes the owner of the file named \ to root:

```
chown root /
```

- If only an owner is given, that user is made the owner of each given file, and the files' group is not changed.
- If the owner is followed by a colon and a group name with no spaces between them, the group ownership of the files is changed as well.

Default Permissions

umask [−S] [mask]

- The **umask** utility sets the file mode creation mask of the current shell execution environment to the value specified by the mask operand.
- **-S** : Accept a symbolic representation of a mask, or return one.
- If no mask is specified, the current umask value is returned.
- The mask shows the bits to be turned off when files are created.

umask Examples

- **umask** 000 : allow read, write, and execute permission for all.
- **umask** 777 : disallow read, write, and execute permission for all.
- **umask** 113 : allow read or write permission to be enabled for the owner and the group, but not execute permission; allow read permission to be enabled for others, but not write or execute permissions.
- **umask** u=rw,go= : allow read and write permission to be enabled for the owner, while prohibiting execute permission from being enabled for the owner; prohibit enabling any permissions for the group and others
- **umask** u-x,g=r,o+w : remove execute permissions for user, set group permissions to read-only, and add write permissions to others.

Links

- A ***link*** is a pointer to a file.
- If two files are linked, then any changes to the data in either file are reflected in the other.
- There a two kinds of links supported by Unix-like operating systems: ***soft/symbolic*** links and ***hard*** links.
- A hard link persists even if the original file is moved, renamed, or deleted, whereas a soft link does not.
- To prevent loops in the filesystem, and to keep the interpretation of .. (parent directory) consistent, many modern operating systems do not allow hard links to directories.

Linking Files

ln [OPTION]… TARGET LINK_NAME

ln [OPTION]… TARGET DIRECTORY

ln [OPTION]… TARGET… DIRECTORY

ln [OPTION]… -t DIRECTORY TARGET…

- In the 1st form, create a link to TARGET with the name LINK_NAME.
- In the 2nd form, create a link to TARGET in the current directory.
- In the 3rd and 4th forms, create links to each TARGET in DIRECTORY.
- Creates hard links by default, symbolic links with **–s, --symbolic**.

Compressing and Archiving Files

- ***Archiving*** combines multiple files into a single file.
- ***Compressing*** reduces the size of a file.
- Archiving then compressing is useful when moving large amounts of data from one system to another e.g. downloading program source files.
- There are many compression (gzip, bzip2, lzma, xz, etc…) and many archiving (tar, ar, cpio, dar) tools.

Compressing Files

gzip [OPTION]… [name …]

bzip2 [OPTION]… [name …]

- **gzip** reduces the size of the named files using using Lempel-Ziv coding.
- **bzip2** compresses files using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding.
- Each file is replaced by a compressed version of itself with the name "original_name.gz" and "original_name.bz2" respectively.
- Files compressed using **bzip2** are generally considerably smaller than files compressed using **gzip**.
- **-d** : decompress file

Archiving Files

tar [-c | -x] [-f ARCHIVE]… [OPTIONS] [ARG…]

- **tar** is used to store multiple files in a single file (an ***archive***), and to manipulate such archives.
- **-c** : Creates an archive e.g.

```
tar –c –f archive.tar foo bar
```

creates an archive file named archive.tar containing the files foo and bar.

- **-x** : Extract files from an archive e.g.

```
tar –x –f archive.tar bar
```

extracts the file bar from the file archive.tar

Finding Files

find [OPTIONS] [starting-point…] [expression]

- **find** searches the directory tree rotted at each given starting-point by evaluating the given expression until the outcome is known at which point **find** moves on to the next file name e.g. to search for files named foo in the home directory, use

```
find /home –name foo
```

- You can search by name, type, size, group etc.

locate [OPTION] PATTERN …

- **locate** reads one or more databases prepared by **updatedb** and writes file names matching at least one of the PATTERNs to standard output, one per line.
- **updatedb** creates or updates a database used by **locate**, and is usually run daily by cron to update the default database.
- **locate** is usually faster than **find**, but can name files that no longer exist and will fail to name files that were created since the last database update.

Finding Commands

which [a-] filename …

- **which** returns the pathnames of the files (or links) which would be executed in the current environment, had its arguments been given as commands in a strictly POSIX-conformant shell.
- It does this by searching the PATH for executable files matching the names of the arguments.

whereis [OPTIONS] [-**BMS** directory… -**f**] name…

- **whereis** locates in the named directory the binary, source and manual files for the specified command names e.g.

```
whereis -bm ls tr -m gcc
```

searches for "ls" and "tr" binaries and man pages, and for "gcc" man pages only.

Diffing and Patching Files

diff [OPTIONS]… FILES

- Compares FILES line by line.
- The output to `diff –u file1 file2` will have sequences of the form:

```
@@ -64,22 +55, 15 @@
            …
-   [A line in file1 to be deleted]
            …
    [A line in file1 to stay the same]
            …
+   [A line in file2 to be added]
```

- **-64,22 +55, 15** means that if the 22 lines in file1 starting from line number 64 are changed using the changes described then they will match the 15 lines in file2 starting from line 55.
- **-r**: recursively compare any subdirectories found.

Diffing and Patching Files

patch [OPTIONS]… file

- The **patch** utility reads a source (patch) file containing any of four forms of difference (diff) listings produced by the **diff** utility (normal, copied context, unified context, or in the style of ed) and apply those differences to a file e.g.

```
diff -u file1 file2 | patch
```

applies a patch to file1 using the output of the diff applied to file1 and file2.

- If two users each have a copy of the same file and one user updates their copy, then sending the patch file to the second user may be more efficient than sending them the new file in case the original file is big but the set of changes is small.
- When more advanced diffs are used (e.g. using the **–u** flag with the diff utility), patches can be applied even to files that have been modified in the meantime, as long as those modifications do not interfere with the patch.

Manipulating Text

Searching for Text

grep [OPTION…] PATTERNS [FILE…]

grep [OPTION…] **-e** PATTERNS … [FILE…]

grep [OPTION…] **-f** PATTERN_FILE [FILE…]

- **grep** searches for PATTERNS in each file and prints each line that matches a pattern.
- **grep** understands three different versions of regular expression syntax: "basic" (BRE), "extended" (ERE) and "perl" (PCRE).
- **-f**: obtain patters from FILE, one per line'
- **-i**: ignore case
- **-r**: recursively read all files under each directory

Searching

If `cat file.txt` outputs

```
Line with banana
Line with orange
Another line with banana
```

then `grep "banana" file.txt` outputs

```
Line with banana
Another line with banana
```

Sorting

sort [OPTION]… [FILES]…

- Write sorted concatenation of all FILE(s) to standard output e.g. if `cat file.txt` outputs

```
bananas
melons
apples
```

then `sort file.txt` outputs

```
apples
bananas
melons
```

Removing Duplicates

uniq [OPTION]… [INPUT [OUTPUT]]…

- Filter adjacent matching lines from INPUT (or standard input), writing to OUTPUT (or standard output) e.g. if `cat file.txt` outputs

```
bananas
bananas
melons
melons
bananas
```

then `uniq file.txt` outputs

```
bananas
melons
bananas
```

Shuffling

shuf [OPTION]… [FILE]…

- Write a random permutation of the input lines to standard output e.g. if `cat file.txt` outputs

```
bananas
melons
apples
```

then `shuf file.txt` may output

```
melons
bananas
apples
```

or any of the other 5 possible permutations of the lines of file.txt

Translating Characters

tr [OPTION]… [SET] [SET]

- Translate, squeeze, and/or delete characters from standard input, writing to standard output e.g.

```
tr 'AEIOU' 'aeiou'
```

translates "QUEUEING" to "QueueiGN".

- SETS are specified as strings of characters, with some sequences interpreted:
  - CHAR1-CHAR2 : all characters from CHAR1 to CHAR2 in ascending order
  - [CHAR*] : in SET2, all copies of CHAR until length of SET1
  - [CHAR*REPEAT] : REPEAT copies of CHAR, REPEAT octal if starting with 0
  - [:alnum:] - all letters and digits
  - [:alpha:] – all letters
  - [:lower:] - all lower case letters etc.

Cutting

cut OPTION… [FILE]…

- Print selected parts of lines from each FILE to standard output.
- **-b**: select only these bytes
- **-c**: select only these characters
- **-f**: select only these fields
- **-d**: specifies a delimiter (tab by default)
- **-s**: suppresses line if delimiter is not found

Cutting

If `cat file.txt` gives the output

```
Alice:658-955-2156:63 First St, Philadelphia, PA:19104
Bob:855-332-8410:485 N Second St, New York, NY,:10001
Eve:425-752-6480:1 Third Av, Los Angeles, CA:90024
ZZZZZZZZZZZZZZZZZZZZZZZZZ
```

then

- `cut –d : -f 1 –s` file.txt prints the names.
- `cut –d : -f 2,3 –s` file.txt prints the phone numbers and addresses.
- `cut -c 1` prints the first letter of the name plus the first character of the last line i.e. Z

Pasting

paste [OPTION]… [FILE]…

- Write lines consisting of the sequentially corresponding lines from each FILE, separated by TABS, to standard output e.g. if `cat file1.txt` outputs

```
Philadelphia
New York
California
```

And `cat file2.txt` outputs

```
PA
NY
CA
```

then `paste –d ', ' file1.txt file2.txt` outputs

```
Philadelphia, PA
New York, NY
California, CA
```

Joining

join [OPTION]… FILE1 FILE2

- For each pair of input lines with identical join fields, write a line to standard output. The default join field is the first, delimited by blanks e.g. if `cat file1.txt` outputs

```
Alice PA
Bob NY
Eve CA
```

and `cat file2.txt` outputs

```
Alice 19104
Eve 90024
```

then `join -1 file1.txt file2.txt` outputs

```
Alice PA 19104
Eve CA 90024
```

Comparing

comm [OPTION]… FILE1 FILE2

- Compare sorted files FILE1 and FILE2 line by line.
- With no options, produce three-column output. Column one contains lines unique to FILE1, column two contains lines unique to FILE2, and column three contains lines common to both files.

Comparing

If `cat file1.txt` outputs

```
apple
banana
melon
```

and `cat file2.txt` outputs

```
apple
banana
mango
```

then `comm file1.txt file2.txt` outputs

```
melon
                apple
                banana
        mango
```

Counting Words

wc [OPTION]… [FILE]…

- Print newline, word, and byte counts for each FILE, and a total line if more than one FILE is specified. A word is a non-zero-length sequence of characters delimited by white space e.g. if `cat file.txt` outputs

```
apple banana
melon
mango
```

Then `wc file.txt` outputs

```
3 4 25 file.txt
```

Adding Line Numbers

nl [OPTION]… [FILE]…

- Write each FILE to standard output, with line numbers added e.g. if `cat file.txt` outputs

```
apple
banana
melon
```

Then `wc file.txt` outputs

```
1 apple
2 banana
3 melon
```

CIS 191

Linux and Unix Skills

LECTURE 3

Lecture Plan

1. Piping and Redirection
2. Processes and Jobs
3. SSH Tools

Piping and Redirection

Standard Streams

- Standard streams are preconnected input and output communication channels between a computer program and its environment when it begins execution.
- The three standard I/O connections are called ***standard input*** (***stdin***), ***standard output*** (***stdout***) and ***standard error*** (***stderr***)
- When a command is executed in the shell, input and output streams are typically connected to the keyboard and terminal display respectively, but this can be changed with ***redirection*** or a ***pipeline***.
- Standard error is another output stream typically used by programs to output error messages or diagnostics.

Piping

- A ***pipeline*** is a set of commands chained together by their standard streams, so that the output text of each process is passed directly as input to the next one.
- `command1 | command2` executes command1, using its output as the input for command2 e.g. the command

```
sort record.txt | uniq | wc
```

evaluates to the number of unique lines in record.txt.

- A ***filter*** is a program that gets its data from stdin and writes its output to stdout.
- Filters combined with pipelining enable you to write complex commands in a simple way.

Redirecting stdin and stdout

- To redirect standard input, use the < operator e.g.

```
tr 'A-Z' 'a-z' < file.txt
```

reads file.txt (as opposed to standard input), converts uppercase characters to lowercase and then prints the result to standard output.

- To redirect standard output, use the > operator e.g.

```
ls -al > file.txt
```

writes the output of `ls -al` in file.txt (as opposed to standard output).

- To append standard output rather than overwrite, use >> instead.

Redirecting stderr

- To redirect standard error, use the > operator and specify the stream by number 2 e.g. if there is no directory named documents in the current directory, then

```
ls documents 2> file.txt
```

writes “ls: cannot access ‘documents': No such file or directory” in file.txt (as opposed to standard error).

- To redirect both standard output and standard error, use the &> operator e.g. if there is a directory named documents1 but no directory named documents2 in the current directory, then.

```
ls documents1 documents2 &> file.txt
```

writes both the error message produced by `ls documents2` as well as the result of `ls documents1` to file.txt.

- To append both standard output and standard use &>> instead.

CAUTION!

- Reading from and writing to the same file in pipelines or when using redirections leads to files being truncated (i.e. wiped clean) e.g.

```
tr ‘a’ ‘A’ < file1 > file1
```

truncates file1 before the **tr** command is executed.

- To get around this use a temporary file e.g.

```
tr 'a' 'A' < file1 > file2 ; cat file2 > file1 ; rm file2
```

tee

tee [OPTION]… [FILE]…

- Read from standard input and write to both standard output and one or more files e.g.

```
ls -l | tee file.txt
```

displays the output of the command `ls -l` and at the same time writes a copy of the output in the file.txt

- **tee** is named after the T-splitter used in plumbing.

xargs

xargs [options] [command [initial-arguments]]

- **xargs** reads items from the standard input, delimited by blanks or newlines, and executes the command one or more times with any initial arguments followed by items read from standard input e.g. the following command deletes all files older than two weeks in the /tmp folder:

```
find /tmp -mtime +14 | xargs rm -r
```

/dev/null

- ***/dev/null*** is a file that discards all data written to it but reports that the write operation succeeded e.g.

```
rm documents 2> /dev/null
```

ignores any errors reported when trying to delete the file nmaed 'documents' from the current directory.

- /dev/null is useful for disposing of unwanted output streams of a process, or as a convenient empty file for input streams.

Processes and Jobs

Processes

- A ***process*** is an instance of a running program.
- Unix is a ***multitasking*** system i.e. it runs several processes at a time using a timesharing system.
- Every active process has a unique Process Identification Number (PID).
- Processes can create processes e.g. whenever a command is issued in the terminal, the process associated with the terminal starts a new process to execute the command.
- Processes can also stop, resume and terminate other processes.
- Look around in the ***/proc*** directory to find out all kinds of things about the currently running processes!

Processes

ps [options]

- Displays information about a selection of the active processes.
- **-e** : Select all processes.
- **-H** : Show process hierarchy.
- **-u** : Select by effective user ID or name.
- **-g** : Select by session or by effective group name.

top [options]

- The **top** program provides a dynamic real-time view of a running system.
- It can display system summary information as well as a list of processes or threads currently being managed by the Linux kernel.

pstree [options]

- **pstree** shows running processes as a tree.

Signaling Processes

kill [-s sigspec] [-n signum] [-sigspec] jobspec or pid

- Send a signal specified by sigspec or signum to the process named by job specification jobspec or process ID pid.
- If sigspec and signum are not present, SIGTERM is used.
- **-SIGTERM** : Terminates the process.
- **-SIGSTOP** : Stops the process.
- **-SIGCONT** : Continues a process if it is stopped.
- **-SIGHUP** : Notifies process that a hang-up has been detected on controlling terminal or that controlling process has a died.
- **-SIGKILL** : Similar to SIGTERM but cannot be caught, blocked or ignored.
- **-SIGQUIT** : Notifies process that user sent a quit signal e.g. with CTRL-D

Prioritizing Processes

nice [OPTION] [COMMAND [ARG]…]

- Run COMMAND with an adjusted niceness, which affects process scheduling.
- Niceness values range from -20 (most favourable to the process) to 19 (least favourable to the process).
- **-n**, **--adjustment**=N : add integer N to the niceness (default 10)
- The NI column of **top** shows the niceness values of currently running processes.

renice [-n] priority [-g|-p|-u] identifier…

- **renice** alters the scheduling priority of one or more running processes.
- The first argument is the priority value to be used, and the other arguments are interpreted as process IDs (by default), process group IDs, user IDs, or user names e.g. the following command changes the priority of the processes with PIDs 987 and 32, plus all processes owned by the users daemon and root:

```
renice +1 987 -u daemon root -p 32
```

Job Control

- ***Job control*** allows a user to start processes in the background, send already running processes into the background, bring background processes into the foreground, and suspend or terminate processes.
- Job control enables a user to regain control of the shell when running programs that either run indefinitely or for long periods of time e.g. moving large quantities of files, compiling source code, playing music etc.

Job Specifications

- The character '%' introduces a ***job specification***, or ***jobspec***.
  - '%n' refers to the job number n.
  - '%%', '%+' and '%' refer to the current job i.e. the last job stopped while it was in the foreground or started in the background.
  - '%-' refers to the previous job.
  - '%str' refers to refer to a job which was started by a command beginning with str; it is an error if there is more than one such job.
  - '%?str' refers to a job that was started by a command containing str; it is an error if there is more than one such job.

Job Control Commands

jobs [OPTION] [jobspec…]

- Displays the status of jobs in the current session.

<command> **&**

- Run the specified command in the background.

bg [jobspec]

- Resume each suspended job jobspec in the background, as if it had been started with '&'.

fg [jobspec]

- Resume the job jobspec in the foreground and make it the current job.

- Type CTRL-Z to stop a currently running process and return control to Bash.
- Type CTRL-C to terminate a currently running process and return control to Bash.

Waiting for Processes

wait [-fn] [jobspec or pid …]

- Wait until the child process specified by each process ID pid or job specification jobspec exits and return the exit status of the last command waited for.
- If a job spec is given, all processes in the job are waited for.
- If no arguments are given, all currently active child processes are waited for, and the return status is zero.
- If the **-n** option is supplied, **wait** waits for a single job to terminate and returns its exit status.
- Supplying the **-f** option, when job control is enabled, forces **wait** to wait for each pid or jobspec to terminate before returning its status, instead of returning when it changes status.

Daemons

- A ***daemon*** is a process that runs in the background and supervises the system or provides functionality to other processes.
- init : The Unix daemon that spawns all other processes.
- systemd : The Linux equivalent of init as of 2016.
- crond : Time-based job scheduler.
- httpd : Web server daemon.
- sshd : Listens for secure shell requests from clients etc…

crond

- ***crond*** is a daemon to execute scheduled commands.
- crond wakes up once a minute, checks to see if there are any tasks scheduled for that minute, and executes them.
- There is no need to ever run the command crond; instead it will be launched at start-up.
- It responds to the presence of crontab files in /etc/crontab.
- /etc/crontab is a system-wide file, but each user can have their own crontab.

crontab

- Entries in crontab files have the following structure:

m h dom mon dow command

where **m** is the minute, **h** is the hour, **dom** is the day of the month, **mon** is the month, **dow** is the day of the week, and **command** is the command to run e.g.

```
0 17 * * sun,fri  /home/smaina/script.sh
```

runs the shell script named "script.sh" in the home directory of the user "smaina" on each Sunday and Friday at 5 PM.

crontab

crontab [-u user] <file | ->

crontab [-u user] <-l | -r | -e> [-i] [-s]

- **crontab** is the program used to install a crontab table file, remove or list the existing tables used to serve the crond daemon.
- Each user can have their own crontab.
- **-u** : specifies the name of the user whose crontab is to be modified.
- **-l** : displays the current crontab on standard input.
- **-e** : edits the current crontab using the editor specified by the VISUAL or EDITOR environment variables.

SSH Tools

SSH

- ***Secure Shell*** (***SSH***) is a cryptographic network protocol for operating network services securely over an unsecured network.
- SSH provides a secure channel over an unsecured network in a client-server architecture, connecting an SSH client with an SSH server.
- The SSH client drives the connection setup process and uses ***public key cryptography*** to verify the identity of the SSH server.
- After the setup phase the SSH protocol uses strong symmetric encryption and hashing algorithms to ensure the privacy and integrity of the data that is exchanged between the client and server.

Public Key Cryptography

- Public key cryptography enables the encryption and decryption of messages over an unsecure channel.
- Every pair of communicating agents has a ***public key*** and a ***private key***.
- Public keys can be viewed by anyone over the channel whereas each private key is known only to the agent that owns it.
- For agents A and B to communicate, A needs to know B's public key, and B needs to know A's public key.
- Agent A uses Agent B's public key to encrypt a message to agent B.
- Agent B then uses their private key to decrypt messages they receive.
- Getting a private key from a public key is hard, which makes decryption practically impossible even though public keys can be viewed by anyone over the channel.

SSH Keys

ssh-agent

- **ssh-agent** is a program that holds private keys used for public key authentication.

ssh-keygen

- **ssh-keygen** generates, manages and converts authentication keys for ssh.
- Normally this program generates the key and asks for a file in which to store the private key.
- The public key is stored in a file with the same name but ".pub" appended.

ssh-add

- **ssh-add** adds private key identities to the authentication agent, **ssh-agent**.
- When run without arguments, it adds the files ~/.ssh/id_rsa, ~/.ssh/id_dsa, ~/.ssh/id_ecdsa, and ~/.ssh/id_ed25519.

SSH Uses

- Logging in to a shell on a remote host.
- Executing a single command on a remote host.
- Backing up, copying, and mirroring files.
- Browsing the web.
- Securing file transfer protocols.
- etc…

The OpenSSH Client

ssh [option] destination [command]

- **ssh** connects and logs into the specified destination, which may be specified as either [user@]hostname or a URI of the form ssh://[user@]hostname[:port] e.g. the following command logs in to the home directory of the user account smaina at the host eniac.seas.upenn.edu:

```
ssh smaina@eniac.seas.upenn.edu
```

- If a command is specified, it is executed on the remote host instead of a login shell e.g. the following command creates a directory named 'foo' in the home directory of the user account smaina at the host eniac.seas.upenn.edu:

```
ssh smaina@eniac.seas.upenn.edu "mkdir foo"
```

- **ssh** is a filter and can therefore be used in pipelines e.g.

```
sort file.txt | ssh smaina@eniac.seas.upenn.edu "uniq" | wc
```

Working with Files Remotely Using SSH

scp [option] source target (GOOD)

- **scp** uses SSH to copy files between hosts on a network e.g. the command

```
scp file.txt smaina@eniac.seas.upenn.edu:~/Documents/
```

copies "file.txt" to the directory "~/Documents" of the user "smaina" on "eniac.seas.upenn.edu".

sftp [option] destination (BETTER)

- **sftp** allows a user to transfer files to and from a remote network site using SSH.

sshfs [user@]host:[dir] mountpoint [options] (BEST)

- **sshfs** (Secure SHell FileSystem) is a file system capable of operating on files on a remote computer using just an SSH login on the remote computer e.g. the command

```
sshfs smaina@eniac.seas.upenn.edu: temp
```

mounts the filesystem found in the home address of the user named "smaina" at the directory named "temp", allowing the user to work with remote filesystem as though it exists locally, and the command

```
fusermount -u temp
```

unmounts the remote filesystem from the "temp" directory.

CIS 191

Linux and Unix Skills

LECTURE 4

Lecture Plan

1. Patterns and Regular Expressions
2. sed
3. AWK

Patterns and Regular Expressions

Patterns and Regular Expressions

- ***Patterns*** and ***regular expressions*** provide a compact notation for describing sets of strings
- Many tools use patterns or regular expressions to match strings e.g. **less**, **more**, **grep**, **find**, text editors such as **Vim** and **Emacs** etc.
- There are many kinds of patterns, and many kinds of regular expressions e.g. ***POSIX basic regular expressions*** (BRE), ***POSIX extended regular expressions*** (ERE), ***PERL regular expressions*** (PCRE) etc.

Filename Expansion/Globbing

- In ***filename expansion***/***globbing***, the shell scans each word on the command line for the characters '*', '?', and '[', and if one of these characters appears, then the word is regarded as a pattern, and replaced with an alphabetically sorted list of filenames matching the pattern.
- ? (question mark) matches a single character e.g.
  - hd? matches only strings with three characters that start with "hd".
- * (asterisk) matches any string including the empty string e.g.
  - m*e matches only that strings that start with "m" and end with "e"
- […] (square brackets) matches any one of the enclosed characters., except if the first character following the '[' is a '!' or a '^' in which case any character not enclosed is matched e.g.
  - [AE]xkd matches only strings that start with of "A" or "E" and end with "xkd"
  - [^BX]y matches only strings that do not start with of "B" or "X" and end with "y"
- \ (backslash) escapes the following character; the escaping backslash is discarded when matching.
- The special pattern characters must be quoted if they are to be matched literally.

Brace Expansion

- ***Brace expansion*** is a mechanism by which arbitrary strings may be generated.
- Patterns to be brace expanded take the form of an optional preamble, followed by either a series of comma-separated strings or a sequence expression between a pair of braces, followed by an optional postscript.
- The preamble is prefixed to each string contained within the braces, and the postscript is then appended to each resulting string, expanding left to right.
- Brace expansions may be nested.
- The results of each expanded string are not sorted; left to right order is preserved e.g.

```
> echo a{b,{1..5},d}e
abe a1e a2e a3e a4e a5e ade
```

POSIX Basic Regular Expressions

- A bracket expression [ ] matches a single character that is contained within the brackets e.g.
  - [abc] matches "a", "b", or "c",
  - [a-z] specifies a range that matches any lowercase letter from "a" to "z",
  - [abcx-z] matches "a", "b", "c", "x", "y", or "z", as does [a-cx-z].
  - [-abc] and [abc-] each match "a", "b", "c", and "-".
- A caret inside a bracket expression [^] matches a single character that is not contained within the brackets e.g.
  - [^abc] matches any character not equal to "a", "b", or "c",
  - [^a-z] matches any single character that is not a lowercase letter from "a" to "z",
  - [^abcx-z] matches any character other than "a", "b", "c", "x", "y", or "z".
  - [^-abc] and [^abc-] each match any character other than "a", "b", "c", or "-".

POSIX Basic Regular Expressions

- A dot '.' matches any single character, except if inside a bracket expression, in which case . is matched literally e.g.
  - a.c matches "abc" but [a.c] matches only "a", ".", or "c".,
  - [a.c] matches only "a", ".", or "c".
- ^ matches the starting position within the string if it is the first character of the regular expression e.g.
  - ^water matches any string that begins with "water".
- $ matches the ending position within the string if it is the last character of the regular expression e.g.
  - melon$ matches any string that ends with "melon".
- * matches the preceding element zero or more times e.g.
  - ab*c matches "ac", "abc", "abbbc", and so on,
  - [xyz]* matches "", "x", "y", "z", "zx", "zyx", "xyzzy", and so on.

POSIX Basic Regular Expressions

- \{m\} matches the preceding element exactly m times e.g.
  - 'a\{3\}' matches only "aaa".
- \{m,\} matches the preceding element at least m times e.g.
  - 'a\{3,\}' matches "aaa", "aaaa", "aaaaa", "aaaaaa", "aaaaaaa", and so on.
- \{m,n\} matches the preceding element at least m and not more than n times e.g.
  - 'a\{3,5\}' matches only "aaa", "aaaa", and "aaaaa".
- \(\) defines a subexpression that is treated as a single element e.g.
  - 'ab*a matches "a", "ab", "abb" and so on,
  - '\(ab\)*' matches "", "ab", "abab", "ababab", and so on.
- \n matches what the nth marked subexpression matched, where n is a digit from 1 to 9 e.g.
  - \2 matches "cd" in \(ab\)\(cd\)

POSIX Basic Regular Expressions

- \+ matches the preceding element one or more times e.g.
  - ab\+c matches "abc", "abbbc", and so on, but not "ac",
  - [xyz]\+ matches "x", "y", "z", "zx", "zyx", "xyzzy", and so on
  - \(ab\)\+ matches "ab", "abab", "ababab", and so on.
- \? matches the preceding element one or zero times e.g.
  - ab\?c matches either "ac" or "abc",
  - \(ab\)\? matches "" or "ab".
- \| matches the preceding element or the following element e.g.
  - \(abc\)\|\(def\) matches either "abc" or "def".

POSIX Extended Regular Expressions

- The main way in which extended regular expression differ from basic regular expressions is that some backslashes are removed: \(…\) becomes (…), \+ becomes +, \? becomes ?, \| becomes |, \{m,n\} becomes {m,n}, \{m\} becomes {m}, \{m,\} becomes {m,}, and \{,n\} becomes {,n}.
- In ERE syntax, the metacharacters (,), [,], . , *, ?, +, |, ^ and $ have to be escaped with a backslash symbol in order to be treated as literal characters e.g. a\.(\(|\)) matches the string "a.(" or "a.)"
- Also, ERE syntax does not support \n for backreferences.

POSIX Character Classes		
POSIX Class	**Similar To**	**Meaning**
[[:upper:]]	[A-Z]	Uppercase letters
[[:lower:]]	[a-z]	Lowercase letters
[[:alpha:]]	[A-Za-z]	Upper- and lowercase letters
[[:digit:]]	[0-9]	Digits
[[:xdigit:]]	[0-9A-Fa-f]	Hexadecimal digits
[[:alnum:]]	[A-Za-z0-9]	Digits, upper- and lowercase letters
[[:punct:]]		Punctuation (all graphic characters except letters and digits)
[[:blank:]]	[\t]	Space and TAB characters only
[[:space:]]	[\t\n\r\f\v]	Blank (whitespace) characters
[[:cntrl:]]		Control Characters
[[:graph:]]	[^ \t\n\r\f\v]	Graphic characters (all characters which have graphic representation)
[[:print:]]	[^\t\n\r\f\v]	Graphic characters and space

sed

sed

sed [OPTION]... {script-only-if-no-other-script} [input-file]...

- **sed** is a stream editor used to perform basic text transformations on an input stream (a file or input from a pipeline).
- **sed** is similar to **tr** but is much more powerful; indeed **sed** is Turing complete.
- **sed** has many commands, with the s command (as in substitute) the most important.
- The syntax of the s command is 's/regexp/replacement/[flags]'.
- The s command attempts to match the pattern space against the supplied regular expression regexp; if the match is successful, then that portion of the pattern space which was matched is replaced with replacement e.g.

```
sed 's/not guilty/guilty/g' file.txt
```

replaces all occurrences of the string "not guilty" with "guilty" in file.txt and sends results to stdout.

The s command

- The replacement can contain \n (n being a number from 1 to 9, inclusive) references, which refer to the portion of the match which is contained between the nth \(and its matching \) e.g.

```
echo "24-09-2007-picturename.jpg" |\
sed -r 's/([0-9]+)-([0-9]+)-([0-9]+)/\3-\2-\1/'
```

prints "2007-09-24-picturename.jpg" to standard output (the option –r uses EREs).

- Also, the replacement can contain unescaped '&' characters which reference the whole matched portion e.g.

```
sed –r 's/[a-z]+/"&"/g'
```

replaces lowercase words with lowercase words in quotes

s command flags

- The s command can be followed by zero or more flags.
- With no flag, the s command will only do one substitution per line.
- **g** (for global) applies the replacement to all matches to the regexp, not just the first.
- **number** replaces the **number**th match of the regexp.
- **p** prints the new pattern space if the substitution was made.
  - With no options, a line with a substitution is printed twice and a line without a substitution is printed once.
  - With the option **–n**, a line with a substitution is printed once and a line without a substitution is not printed at all.
- **i** and **I** match regexp in a case-insensitive manner.

Specifying Lines

- To restrict a command to line number n just add n before the command e.g. to restrict a command to line 3 use:

```
sed '3 s/[a-z]\+/"&"/g'
```

- To restrict a command to a range of line numbers add the two numbers before the command separated by a comma e.g. to restrict a command to lines 23 to 54 use:

```
sed '23,54 s/[a-z]\+/"&"/g'
```

and to restrict from the 50th line to the end of the file use:

```
sed '50,$ s/[a-z]\+/"&"/g'
```

Specifying Lines

- To restrict a command to lines that match a regular expression add that regular expression before the command e.g. to restrict to lines that start with a "#" use:

```
sed '/^#/s/[a-z]\+/"&"/g'
```

- To restrict a command to a range specified by two regular expressions add the two regular expressions separated by a comma before the command e.g. to restrict a command to each line that lies between a line starting with a "#" and a line ending with a "%" use:

```
sed '/^#/,/%$/s/[a-z]\+/"&"/g'
```

Enough sed?

- **sed** has many more features!
- To learn more about **sed**, read this (relatively) easy tutorial:

  http://www.grymoire.com/Unix/Sed.html

- This **sed** script archive has lots of interesting and clever examples of **sed** scripts:

  http://sed.sourceforge.net/grabbag/scripts/

AWK

AWK

- AWK is a programming language for text processing and typically used as a data extraction and reporting tool.
- AWK was created at Bell Labs in the 1970s and its name is derived from the surnames of its authors – Alfred Aho, Peter Weinberger, and Brian Kernighan.
- Many implementations of AWK exist : gawk (GNU awk), mawk, nawl, tawk, etc.

AWK Overview

- AWK is designed for processing text files.
- AWK treats a file as a sequence of ***records***; by default each line is a record.
- Each line is broken up into a sequence of ***fields***; the first word in a line is the first field, the second word is the second field, and so on.
- An AWK program is a sequence of ***pattern-action statements***.
- AWK reads the input a line at a time.
- A line is scanned for each pattern in the program, and for each pattern that matches, the associated action is executed.

AWK Example

- The AWK program:

```
BEGIN       {print "START"}
/^[0-9]/    {print}
/melons$/   {print "Found melons!"}
END         {print "STOP"}
```

prints "START", and, for each line in the input line by line, prints the line if starts with a digit, prints "Found melons!" if the line ends with "melons", and finally prints "STOP".

AWK

- AWK is a similar to many programming languages in that it has the following features:
  1. Structured control flow i.e. if, while, assignment and looping statements,
  2. Arithmetic and Boolean expressions,
  3. Built-in arithmetic and string functions.
  4. Arrays,
  5. User-defined functions.
- AWK is different from other programming languages in that has several built-in variables for easy processing of text.

AWK Fields

- AWK automatically separates each input line into fields, each referred to by a number.
- $0 refers to the whole line
- $1, $2, …, $9, $(10), … refer to each field.
- The default Field Separator (FS) is white space.

AWK Built-in Variables

- NR : Keeps a current count of the number of input records.
- NF : Keeps a count of the number of fields in an input record.
  - The last field in the input record can be designated by $NF.
- FILENAME : Contains the name of the current input-file.
- FS : Contains the "field separator" character used to divide fields on the input record, by default any whitespace character.

AWK Built-in Variables

- RS : Stores the current "record separator" character, by default a "newline".
- OFS : Stores the "output field separator", which separates the fields when AWK prints them, by default a "space" character.
- ORS: Stores the "output record separator", which separates the output records when AWK prints them, by default a "newline" character.
- OFMT : Stores the format for numeric output, by default "%.6g".

The print command

- The print command is used to output text.
- 'print' displays the contents of the current record.
- 'print $1' displays the first field in the current record.
- 'print $1 $3' concatenates the first and third fields of the current record and displays the result.
- 'print $1, $3' displays the first and third fields of the current record, separated by the output field separator (OFS).
- 'print "expression" > "file name"' sends output to a file.
- 'print "expression" | "command"' pipes output to a command.

Reversing Words in AWK Example

```
{
        for(i=1;i<=NF;i++) {
                tmp = ""
                for(j=length($i);j>0;j--) {
                        char = substr($i,j,1)
                        tmp = tmp char
                }
                $i = tmp
        }
        print
}
```

More AWK?

To learn more about **AWK**, read this (relatively) easy tutorial:

http://www.grymoire.com/Unix/Awk.html

CIS 191

Linux and Unix Skills

LECTURE 5

Lecture Plan

1. Vim
2. Emacs

Text Editors

- There are many text editors available for Unix-like systems: vi, Vim, Emacs, nano, pico, sublime, gedit etc.
- vi is part of the Unix specification and is therefore included in all SUS and POSIX compliant operating systems.
- Historically, Vim and Emacs have enjoyed the most attention from the Unix/Linux community.
- Vim and Emacs both have a steep learning curve but provide greater functionality than most editors in addition to being highly extensible and customizable.

Buffers and Windows

- Every file that you open in Vim or Emacs is associated with a ***buffer***.
- Any changes you make to a file are tracked within the buffer, and the file is only updated once the buffer is written to disk.
- A ***window*** is an area of the screen that allows you to view the contents of a buffer.
- Both Vim and Emacs allow you split the screen into multiple windows that can be viewed at the same time.
- If a buffer is open in different windows, then changes made to the buffer in one window are reflected in other windows at the same time.

Commands and Key-bindings

- In both Vim and Emacs, certain key combinations cause a command to be carried out rather than cause text to get inserted into the active buffer.
- Vim and Emacs do not assign meanings to keys directly; instead, they assign meanings to named commands, and then give keys their meanings by binding them to commands.
- Most Emacs key-bindings are non-modal i.e. most key combinations cause the same command to be executed regardless of the current buffer mode, while Vim key-bindings are modal i.e. a key combination can lead to different effects depending on the current mode of the buffer.
- Learning default key-bindings and how to use them efficiently is one of the most difficult parts of mastering each of these editors!

Vim

Vim

- ***Vim***, short for vi Improved, is a clone, with additions, of Bill Joy's ***vi*** text editor program for Unix.
- Vim is mostly, but not entirely compatible with vi as defined in the Single Unix Specification and POSIX.
- Vim is a modal editor, which means that Vim functions differently depending on the mode that it is in.
- The four main modes are ***normal/command mode***, where keystrokes are interpreted as commands that control the edit session, ***insert mode***, where the text you type is inserted into the buffer, ***visual mode***, where commands apply to a highlighted area, and ***command-line mode***, which is used to enter Ex commands, search patterns and filter commands.

Normal Mode

- In ***normal/command*** mode, keystrokes are interpreted as commands that control the edit session.
- Normal mode commands consist mainly of motion commands that move the cursor to a new position, and editing commands which edit the buffer contents.
- Motion and editing commands can be combined to form complex commands e.g. the command '4dj' combines the motion command 'j' that moves the cursor one line up with the editing command 'd' that cuts text, to cut the current line as well as the 4 lines above it.

Some Commonly Used Motion Commands	
h/j/k/l	Move cursor one line left/down/up/right.
G	Move cursor to the start of the last line of the file.
gg	Move cursor to the start of the first line of the file.
e	Move cursor to the end of the current word.
b	Move cursor to the beginning of the current word.
w	Move cursor to the beginning of the next word.
0	Move cursor to the beginning of the current line.
^	Move cursor to the first non-whitespace character of the current line.
$	Move cursor to the end of the current line.
(	Move cursor to the beginning of the previous sentence.
)	Move cursor to the beginning of next sentence.
{	Move cursor to the beginning of the previous paragraph.
}	Move cursor to the beginning of the next paragraph.
m**a**	Set mark **a** at current cursor position.
`**a**	Move cursor to the position of mark **a**.

Motion Commands

When it applies, motion commands can be modified with numbers to execute the command several times or to a certain point e.g.

- 50G moves the cursor to the start of line 50.
- 3w moves the cursor to the beginning of the 3rd next words.
- 15k moves the cursor 15 lines up.
- 4b moves the cursor to the beginning of the 4th previous word.

Some Commonly Used Editing Commands	
r	Replace a single character ('r' for replace, e.g. 'rw' replaces with 'w').
x	Delete a single character.
d	Cut to modifier e.g. 'dw' cuts to the beginning of the next word ('d' for delete).
dd	Cut line.
y	Copy to modifier e.g. 'yw' copies to the beginning of the next word ('y' for yank).
yy	Copy line.
c	Cut to modifier and enter into insert mode e.g. 'cw' cuts to the beginning of the next word and enters into insert mode.
cc	Cut line and enter into insert mode
p	Paste after cursor.
J	Join line below cursor position to the current one with one space in between.
u	Undo last change.
CTRL-r	Redo last change.
.	Repeat last command.

Modifying Editing Commands

- Motion and editing commands can be combined to form complex commands e.g.
  - db cuts to the beginning of the current word,
  - y$ copies from the current cursor position to the end of the current line,
  - 4dj cuts the current line as well as the 4 lines above it,
  - `ay`b copies the text between positions of marks a and b,
  - 4cw cuts the next four words then switches to insert mode

Switching Modes

- Press the escape key (ESC) or CTRL-[to switch from insert mode to normal mode, or from visual mode to normal mode.
- Press 'v' or 'V' to switch from normal mode to visual mode.
- Press SHIFT-I to switch from visual mode to insert mode.
- There are several ways to switch from normal mode to insert mode:
  - Press 'i' to enter insert mode before the current cursor position.
  - Press 'A' to enter insert mode at the end of the current line.
  - Press 'o' to insert a line below the current line, then enter insert mode.
  - Press 'O' to insert a line above the current line, then enter insert mode.
  - Press 'c' to cut to modifier and enter insert mode e.g. 'ce' cuts to the end of the next word and enters insert mode.

Visual Mode

- Using Visual mode consists of three parts:
  1. Mark the start of the text with 'v', 'V' or CTRL-V. The character under the cursor will be used as the start.
  2. Move to the end of the text. The text from the start of the Visual mode up to and including the character under the cursor is highlighted.
  3. Type an operator command. The highlighted characters will be operated upon.
- 'v' starts visual mode per character, 'V' starts visual mode per line, and CTRL-V starts visual mode per block.

Some Commonly Used Visual Mode Operators	
c	Cut and enter insert mode.
d	Delete.
y	Copy (yank).
~	Swap case.
gu	Make lowercase.
gU	Make uppercase.
!	Filter through an external program (e.g. '!nl' numbers the highlighted lines).
gq	Format text.
>	Shift right.
<	Shift left.

Command-line Mode

Searching	
/pattern	Search for pattern.
?pattern	Search backwards for pattern.
/\<search_string\>	Search for variable “search_string”, ignoring words containing “search_string”.
*	Move cursor to the next instance of the current word.
#	Move cursor to the previous instance of the current word.
n	Move cursor to the next instance of the current word, in the direction specified by the last use of {*,#}.
N	Move cursor to the previous instance of the current word, in the direction specified by the last use of {*,#}.
:set hlsearch	Highlight searched words.
:nohlsearch	Un-highlight searched words.

Replacing

- `:[range]s/R/repl/[flags]` means replace all strings that match R with repl in range according to flags.
- Ranges :
  - % : The entire file.
  - '<,'> : The current selection; the default range while in visual mode.
  - 25 : Line 25.
  - 25,50 : Lines 25-50.
  - $: Last line; can be combined with other lines as in '50,$'.
  - . : Current line; can be combined with other lines as in '.,50'.
  - ,+2 : The current lines and the two lines below.
  - -2, : The current line and the two lines above.
- Flags :
  - g : Replace all occurrences on the specified line(s)
  - i : Ignore case
  - c : Confirm each substitution

Working With Multiple Buffers and Windows

:e somefile	Open 'somefile' in a new buffer ('e' for edit).
:bn	Go to the next buffer.
:bp	Go to the previous buffer.
:bd	Delete a buffer.
:ls	List all open buffers
:split	Split the selected window into two windows, one above the other.
:vsplit	Split the selected window into two windows, one next to the other.
CTRL-w h/j/k/l	Switch to window left/down/up/right.

Saving Files and Exiting Vim	
:w	Save the current file ('w' for write).
:w newname	Save a copy of the current file as 'newname' but continue editing the original file.
:sav newname	Save a copy of the current file as 'newname' and continue editing the file 'newname'.
:wq	Save the current file and closes current buffer ('wq' for write then quit).
:q!	Closes current buffer without saving.
:qa	Closes all buffers.
:x	Like :wq, but write only when changes have been made.

Customizing Vim

- There are many plugins available for Vim, usually written in ***vimscript***, that extend or add new functionality to Vim.
- Vim has a recording feature allows for the creation of ***macros*** to automate sequences of keystrokes and call internal or user-defined functions and mappings.
- Edit your .vimrc file to make Vim the right editing environment for you.

Emacs

Emacs

- ***Emacs***, short for extensible macros, is a family of text editors characterized by their extensibility.
- The original EMACS was written in 1976 by Carl Mikkelsen, David A. Moon and Guy L. Steele Jr.
- GNU Emacs is currently the most used variant of Emacs and was first released by Richard Stallman in 1985.

Emacs Modes

- Emacs contains many editing modes that Mer its basic behaviour in useful ways.
- ***Major modes*** provide specialized facilities for working on a particular file type, such as a C source file, or a particular type of non-file buffer, such as a shell buffer.
- Major modes are mutually exclusive; each buffer has one and only one major mode at any time.
- ***Minor modes*** are optional features which you can turn on or off, not necessarily specific to a type of file or buffer e.g. auto-fill-mode wraps lines automatically when they get longer than 70 characters while flyspell-mode highlights misspelled words as you type.

Some Commonly Used Cursor Movement and Deletion Commands				
Cursor Movement	**Operation**			
	Move		**Delete**	
Amount	**forward**	**backward**	**forward**	**backward**
Character	CTRL-f	CTRL-b	CTRL-d	BACKSPACE
Word	ALT-f	ALT-b	ALT-d	ALT-BACKSPACE
Line	CTRL-n	CTRL-p	CTRL-k (to EOL)	CTRL-SPACE CTRL-a CTRL-w
Sentence	ALT-e	ALT-a	ALT-k	CTRL-x BACKSPACE
Expression	CTRL-ALT-f	CTRL-ALT-b	CTRL-ALT-k	CTRL-ALT-BACKSPACE
Paragraph	ALT-}	ALT-{	N/A	
End/Start of Line	CTRL-e	CTRL-a	N/A	
End/Start of Buffer	ALT->	ALT-<	N/A	

Marking Text	
CTRL-SPACE	Set the mark at point, and activate it
CTRL-x CTRL-x	Set the mark at point, and activate it; then move point where the mark used to be.
CTRL-SPACE CTRL-SPACE	Set the mark, pushing it onto the mark ring, without activating it.
CTRL-u CTRL-SPACE	Move point to where the mark was, and restore the mark from the ring of former marks.

Editing	
CTRL-w	Kill (i.e. cut).
ALT-w	Copy.
CTRL-y	Paste.
ALT-y	Cycle through kill ring (first paste with CLTR-Y, then cycle through kill ring with ALT-Y).
CTRL-_ (or CTRL-x u)	Undo.
CTRL-g CTRL-u	Redo (CTRL-G toggles CTRL-_ from undoing to redoing i.e. press CTRL-_ to undo, then CTRL-G CTRL-_ to redo then CTRL-G CTRL-_ to undo again and so on).
CTRL-x TAB	Indent region.
CTRL-x CTRL-l	Convert to lower case.
CTRL-x CTRL-u	Convert to upper case.

Searching and Replacing	
CTRL-s	Incremental search forward; searches as you type.
CTRL-r	Incremental search backward.
CTRL-s <RET> *string* <RET>	Non-incremental search forward for *string*.
CTRL-s <RET> *string* <RET>	Non-incremental search backward for *string*.
CTRL-ALT-s	Regexp search forward.
CTRL-ALT-r	Regexp search backward.
ALT-x replace-string <RET> string <RET> newstring <RET>	Replace every occurrence of *string* with *newstring*. All occurrences from the current cursor position to the end of buffer are replaced. When a region is active, replacement is limited to the region.
ALT-x replace-regexp <RET> *regexp* <RET> *newstring* <RET>	Ditto but with regexp substituted for string.
ALT-% *string* <RET> *newstring* <RET>	Finds occurrences of string one by one, displays each occurrence and asks you whether to replace it.
CTRL-ALT-% *regexp* <RET> *newstring* <RET>	Ditto but with *regexp* substituted for *string*.

Working with Multiple Buffers and Windows	
CTRL-x CTRL-f *path* <RET>	Find file at *path* and open it in active buffer.
CTRL-x b *buffer*	Select or create a buffer named *buffer*.
CTRL-x LEFT	Select the previous buffer in the buffer list.
CTRL-x RIGHT	Select the next buffer in the buffer list.
CTRL-x CTRL-B	List the existing buffers.
CTRL-x K buffer	Kill buffer buffer.
CTRL-x 2	Split the selected window into two windows, one above the other.
CTRL-x 3	Split the selected window into two windows, positioned side by side.
CTRL-x O	Select another window.
CTRL-x 0	Delete the select window.
CTRL-x 1	Delete all windows in the selected frame except the selected window.

Miscellaneous	
CTRL-x CTRL-x	Save the current buffer to its file.
CTRL-x s	Save any or all buffers to their files.
CTRL-x CTRL-c	Exit Emacs.
CTRL-g	Quit current command.
CTRL-h ?	Open help page in new buffer.
ALT-x shell	Starts a shell in the buffer named *shell*, switching to it if it already exists.
ALT-x compile	Invokes make (with targets and options of your choice) and displays output in a new buffer.

Customizing Emacs

- ***Evil*** is an extensible vi layer for Emacs that provides Vim features like Visual selection and text objects.
- Emacs has many plugins available, usually written in ***emacs lisp*** (elisp), that extend or add new functionality.
- Emacs has a recording feature allows for the creation of ***macros*** to automate sequences of keystrokes and call internal or user-defined functions and mappings.
- Edit your .emacs file to make Emacs the right editing environment for you.

CIS 191

Linux and Unix Skills

LECTURE 6

Lecture Plan

Shell Scripting I

1. Shell Potpourri
2. Shell Parameters
3. Arrays
4. Arithmetic Expansion
5. Conditional Tests

Shell Potpourri

Script Files

- A ***script file*** is a text file containing commands that can be executed by a certain interpreter e.g. a shell script is a file containing shell commands while a sed script is a file containing sed commands.
- The first line of a script file usually begins with '#!' – a shebang – followed by the path to an interpreter.
- When a script starting with the shebang (#!) line:

```
#!/path/to/interpreter
```

is executed, the interpreter found at the location /path/to/interpreter is run, with its first argument being the name of the file e.g. if a script file named "script.sh" in the current directory starts with the line

```
#!/usr/local/bin/bash
```

then the command

```
./script.sh
```

behaves like the command

```
/usr/local/bin/bash script.sh
```

Shell Scripts

- A ***shell script*** is a text file containing shell commands.
- Bash executes the commands in a script file sequentially, line by line.
- Failed script commands do not cause the entire script to fail; instead error messages are often sent to stderror and execution of the script file continues with the next command in the script file.

Exit Statuses

- Every command returns an ***exit status***, sometimes referred to as a ***return status*** or ***exit code***.
- A successful command returns a 0, while an unsuccessful one returns a non-zero value that usually can be interpreted as an ***error code***.
- Likewise, functions within a script and the script itself return an exit status, in which case the last command executed in the function or script determines the exit status.
- Within a script, an **exit n** command exist the shell and returns an exits status of **n** to the shell's parent, where **n** is an integer in the 0 - 255 range.
- When a script ends with an **exit** that has no parameter, the exit status of the script is the exit status of the last command executed in the script.
- The exit status is used by the Bash conditional commands and some of the list constructs.

Exit Statuses

- Exit codes 1 - 2, 126 - 165, and 255 have special meanings, and should therefore be avoided for user-specified exit parameters.
- Exit code 1 is a catch-all for general errors.
- Exit code 2 is reserved for misuse of shell built-ins.
- Exit code 126 means that the command invoked cannot execute.
- Exit code 127 means that the command invoked cannot be found.
- Exit code 128 means that the exit command was invoked with an invalid argument.
- Exit code 128+n means fatal error signal "n".
- Exit code 130 means script terminated by Control-C.
- Exit code 255 means the exit status is out of range.

Special Characters

- Some characters are evaluated by the shell to have a non-literal meaning.
- Instead, these characters are interpreted to carry out a command.
- We can cause the shell to interpret special characters literally by escaping them or placing them in quotes.

Common Special Characters

Character	Description
$	Expansion – Introduce various types of expansion
\	Escape - Prevent the next character from being interpreted as a special character.
#	Comment - Begin a commentary that extends to the end of the line.
=	Assignment - Assign a value to a variable.
{ }	Inline group – Treat the commands inside curly braces as if they are one command.
>, >>, <	Redirection - Redirect a command's output or input to a file.
|	Pipe - Send the output from one command to the input of another command.
*, ?	Globs - Match parts of filenames according to the "wildcard" character.
~	Home Directory
&	Background - Run the command in the background.
;	Command Separator - Separate multiple commands that are on the same line.

Quoting

- Quoting is used to remove the special meaning of certain characters or words to the shell.
- There are three quoting mechanisms: the escape character, single quotes, and double quotes.
- A non-quoted backslash '\' preserves the literal value of the next character that follows, with the exception of newline.
- Enclosing characters in single quotes (''') preserves the literal value of each character within the quotes.
- Enclosing characters in double quotes ('"') preserves the literal value of all characters within the quotes, with the exception of '$', '`', '\', and, when history expansion is enabled, '!'.

Command Substitution

- ***Command substitution*** allows the output of a command to replace the command itself.
- Command substitution occurs when a command is enclosed as follows:

```
$(command) or `command`
```

e.g.

```
echo $(whoami)
```

prints the user name associated with the current effective user ID whereas

```
echo whoami
```

prints the literal string `'whoami'.`

Arithmetic Expansion

- ***Arithmetic expansion*** allows the evaluation of an arithmetic expression and the substitution of the result.
- The format for arithmetic expansion is:

  $((expression))

- **CAUTION**: Bash has only integer math; no floating point math.
- Arithmetic expressions include
  - Binary arithmetic operations : multiplication (*), division (/), remainder (%), addition(+), subtraction (-).
  - Unary arithmetic operations : unary minus (-), unary plus (+).
  - Binary logical operators : bitwise AND (&), bitwise exclusive OR (^), bitwise OR (|), left bitwise shift (<<), right bitwise shift (>>), logical AND (&&), logical OR (||).

Arithmetic Expansion

- Arithmetic expressions include:
  - Unary logical operators : logical negation (!), bitwise negation (~).
  - Comparison operators : less than or equal to (<=), greater than or equal to (>=), less than (<), greater than (>), equal to (==), not equal to (!=).
  - Increment and decrement operators : variable post-increment (id++), : variable post-decrement (id--), : variable pre-increment (++id), : variable pre-decrement (--id).
  - Conditional operator : expr ? expr : expr.
  - Assignment operators : =, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=.
  - List operator : expr, expr (the arithmetic expressions are evaluated from left to right, with the last expression being the result of the expression).

Command Lists

- For a command list of the form C1 && C2, bash executes C1 and then executes C2 if and only if C1 exits with a status of 0.
- For a command list of the form C1 || C2, bash executes C1 and then executes C2 if and only if C1 exits with a status not equal to 0.
- For a command list of the form C1 ; C2, bash executes C1, waits for C1 to exit with any status, and then executes C2.
- Separating C1 and C2 with a newline is equivalent to executing C1 ; C2.
- && and || are left associative e.g. C1 && C2 || C3 = (C1 && C2) || C3.

Shell Parameters

Shell Parameters

- A ***parameter*** is an entity that stores a value.
- A parameter can be a name, a number, or a special character that the shell interprets in a special way.
- A ***variable*** is a parameter denoted by a name.

Shell Variables

- Bash allows you to store ***shell variables*** for the length of time that the shell is open like so:

```
x=3
```

- **CAUTION:** Do not put a space before or after the equals sign!
- Shell variables can be defined to be read-only like so:

```
readonly x=3
```

- Read-only variables cannot be changed by subsequent assignment, nor can they be unset by the **unset** utility.
- To read the values in shell variables, precede their names by a dollar sign ($) e.g.

```
echo "$x"
```

- Shell variables are not available to processes spawned by the shell.

Environment Variables

- ***Environment variables*** are variables that are passed as copies to each process spawned by the shell.
- If an environment variable is changed in a process spawned by a shell, the original variable in the parent shell remains unchanged.
- Environment variables are defined like so:

```
export X=3
```

- The environment for any simple command or function may be augmented temporarily by prefixing it with parameter assignments e.g.

```
X=3 Y=4 ./script.sh
```

Environment Variables

- Some environment variables that you are likely to encounter are:
  - $USER : The current user in the terminal session.
  - $HOME : The current user's home directory.
  - $SHELL : The path to the current command shell.
  - $PATH : A colon-separated list of directories in which the shell looks for commands.
  - $EDITOR : The lightweight program used for editing files.
  - $VISUAL : The full-fledged editor used for editing files, e.g. as Vim or Emacs.

Positional Parameters

- A ***positional parameter*** is a parameter denoted by one or more digits, other than the single digit 0.
- $1, $2, …, $9, ${10}, … are the values of the first, second, … arguments when the script is invoked.
- Positional parameters are assigned from the shell's arguments when it is invoked, and may be reassigned using the **set** built-in command e.g.

```
set c a b
```

sets $1 to c, $2 to a, and $3 to b.

- When a positional parameter consisting of more than a single digit is expanded, it must be enclosed in braces.

Special Parameters

- $* expands to the positional parameters, starting from one.
  - When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by a space i.e. "$*" is equivalent to "$1 $2 … $n".
- $@ expands to the positional parameters, starting from one.
  - When the expansion occurs within double quotes, each parameter expands to a separate word i.e. "$@" is equivalent to "$1" "$2" … "$n".
  - **CAUTION**: You almost always want to use "$@" rather than "$*".
- $# expands to the number of positional parameters.

Special Parameters

- $? expands to the exit status of the most recently executed foreground pipeline.
- $- expands to the current option flags as specified upon invocation.
- $$ expands to the process ID of the shell.
- $! expands to the process ID of the job most recently placed into the background, whether executed as an asynchronous command or using the **bg** builtin.
- $0 expands to the name of the shell or shell script.
- $_ expands to the last argument to the previous simple command executed in the foreground.

Listing, Setting and Unsetting Parameters

env [OPTION]... [-] [NAME=VALUE]... [COMMAND [ARG]...]

- Sets each NAME to VALUE in the environment and runs COMMAND.
- If no COMMAND, print the resulting environment.

set [OPTION] [argument...]

- When arguments are specified, they cause positional parameters to be set or unset.
- If no options or arguments are specified, **set** prints the names and values of all shell variables.

unset [−fv] name...

- unsets values and attributes of variables and functions.
- If **–v** is specified, name refers to a variable name, and if **–f** is specified, name refers to a function.

Parameter Expansion

- The '$' character introduces parameter expansion, command substitution, or arithmetic expansion.
- The parameter name or symbol to be expanded may be enclosed in braces, which are optional but serve to protect the variable to be expanded from characters immediately following it which could be interpreted as part of the name.
- The basic form of ***parameter expansion*** is ${*parameter*} which substitutes the value of *parameter*.
- The braces are required when *parameter* is a positional parameter with more than one digit, or when *parameter* is followed by a character that is not to be interpreted as part of its name.
- If the first character of *parameter* is an exclamation point, bash uses the value formed by expanding the rest of *parameter* as the new *parameter*; this is then expanded and that value is used in the rest of the expansion, rather than the expansion of the original *parameter*.
- **CAUTION:** The exclamation point must immediately follow the left brace in order to introduce indirection.
- **CAUTION**: You should always place any non-literal occurrence of '$' between double quotes!

Parameter Expansion Example

```
> x=One
> echo "ThisIs${x}Var"
ThisIsOneVar
> set a b c d r f g h i j
> echo "${10}"
j
> index=4
> echo "${!index}"
d
```

Substring Expansion

- ${*parameter*:*offset*} expands to the substring of the value of *parameter* starting at the character specified by *offset* and extending to the end of the value.
- ${*parameter*:*offset*:*length*} expands to up to *length* characters of the value of *parameter* starting at the character specified by *offset*.
- If *length* evaluates to a number less than zero, it is interpreted as an offset in characters from the end of the value of *parameter* rather than a number of characters, and the expansion is the characters between *offset* and that result.
- If *offset* evaluates to a number less than zero, the value is used as an offset in characters from the end of the value of *parameter*.
- A negative *offset* must be separated from the colon by at least one space.
- Substring indexing is zero-based unless the positional parameters are used, in which case the indexing starts at 1 by default.

Substring Expansion Examples

```
> string=01234567890abcdefgh
> echo "${string:7}"
7890abcdefgh
> echo "${string:7:2}"
78
> echo "${string:7:-2}"
7890abcdef
> echo "${string: -7:-2}"
bcdef
```

Substring Expanding Positional Parameters

- Substring Expansion can be used with positional parameters if *parameter* is '@', except that an expansion error occurs if *length* evaluates to a number less than zero e.g.

```
> set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f g h
> echo "${@:7}"
7 8 9 0 a b c d e f g h
> echo "${@:7:2}"
7 8
> echo "${@: -7:2}"
b c
> echo "${@:7:-2}"
bash: -2: substring expression < 0
```

Arrays

Arrays

- Bash provides one-dimensional indexed and associative array variables.
- Indexed arrays are referenced using integers (including arithmetic expressions) and are zero-based, while associative arrays use arbitrary strings.
- An indexed array is created automatically if any variable is assigned to using the syntax:

```
name[subscript]=value
```

- **CAUTION**: Do not declared an associative array like this!
- To explicitly declare an indexed array, use:

```
declare -a name
```

- Associative arrays are created using:

```
declare -A name
```

Arrays

- Arrays are assigned to using compound assignments of the form:

```
name=(value1 value2 … )
```

where each value is of the form '[*subscript*]=*string*'.

- Indexed array assignments require only *string*, but when assigning to an associative array, the *subscript* is required.
- When assigning to an indexed array, negative indices count back from the end of the array, and an index of -1 references the last element.

Arrays

- Any element of an array may be referenced using ${*name*[*subscript*]}.
- ${#*name*[*subscript*]} expands to the length of ${*name*[*subscript*]}, and if *subscript* is '@' or '*', the expansion is the number of elements in the array.
- "${*name*[*]}" expands to a single word with the value of each array member separated by a space.
- "${*name*[@]}" expands each element of *name* to a separate word.
- ${!*name*[@]} and ${!*name*[*]} expand to the keys (indices) assigned in array variable *name*.
- **unset** *name*[*subscript*] destroys the array element at index *subscript*.

Substring Expanding Indexed Arrays

- Substring expansion can be used with indexed arrays if *parameter* is an indexed array name subscripted by '@' or '*', except that expansion raises an error if *length* evaluates to a number less than zero:

```
> A=(0 1 2 3 4 5 6 7 8 9 0 a b c d e f g h)
> echo "${A[@]:7}"
7 8 9 0 a b c d e f g h
> echo "${A[@]:7:-2}"
7 8 9 0 a b c d e f
> echo "${A[@]: -7:2}"
b c
> echo "${A[@]: -7:-2}"
bash: -2: substring expression < 0
```

Conditional Tests

Conditional Tests

- Conditional expressions are used by the [[, , test and [, commands e.g.

```
> test 2 -gt 1
> echo "$?"
0
> [ 5 -le 3 ]
> echo "$?"
1
```

- Conditional expressions return an exit status corresponding to the result of the comparison; 0 for true, and 1 for false.
- **CAUTION**: There must be a space after '[' or '[[', before ']' or ']]', as well as before and after the conditional operator.

Conditional Tests

- A conditional test fails if it is evaluated with 0 arguments, e.g.

```
> [ ]
> echo "$?"
> 1
```

- A conditional expression evaluated with 1 argument is true if, and only if, the argument is not null, e.g.

```
> [ "blah" ]
> echo "$?"
> 0
```

Some File Tests

- **-a** *file* or **-e** *file* : True if *file* exists.
- **-r** *file* : True if *file* exists and is readable.
- **-w** *file* : True if *file* exists and is writable.
- **-x** *file* : True if *file* exists and is executable.
- **-d** *file* : True if *file* exists and is a directory.
- **-f** *file* : True if *file* exists and is a regular file.
- **-s** *file* : True if *file* exists and has a size greater than zero.
- *file1* **-nt** *file2* : True if *file1* is newer (according to modification date) than *file2*, or if *file1* exists and *file2* does not.
- *file1* **-ot** *file2* : True if *file1* is older than *file2*, or if *file2* exists and *file1* does not.

String Tests

- **-z** *string* : True if the length of *string* is zero.
- **-n** *string* or *string* : True if the length of *string* is non-zero.
- *string1* **==** *string2* or *string1* **=** *string2* : True if the *strings* are equal.
- *string1* **!=** *string2* : True if the *strings* are not equal.
- *string1* **<** *string2* : True if *string1* sorts before *string2* lexicographically.
- *string1* **>** *string2* : True if *string1* sorts after *string2* lexicographically.

Arithmetic Tests

- *exp1* **-eq** *exp2* : True if *exp1* is equal to *exp2*.
- *exp1* **-ne** *exp2* : True if *exp1* is not equal to *exp2*.
- *exp1* **-lt** *exp2* : True if *exp1* is less than *exp2*.
- *exp1* **-le** *exp2* : True if *exp1* is less than or equal to *exp2*.
- *exp1* **-gt** *exp2* : True if *exp1* is greater than *exp2*.
- *exp1* **-ge** *exp2* : True if *exp1* is greater than *exp2*.

Combining Tests

! *expr* : True if *expr* is false.

(*expr*) : Returns the value of *expr*.

expr1 **-a** *expr2* : True if both *expr1* and *expr2* are true.

expr1 **-o** *expr2* : True if either *expr1* or *expr2* is true.

[[...]] vs [...]

- [[is bash-specific, though others shells may also implement similar constructs.
- No filename expansion or word splitting takes place between [[and]].
  - For example, with [you have to write

```
[ -f "$file" ]
```

    to correctly handle empty strings or file names with spaces in them, but with [[the quotes are unnecessary:

```
[[ -f $file ]]
```

- With [, '==' and '!=' are literal string comparisons, while with [[, '==' and '!=' apply pattern matching rules.
  - For example, the following test succeeds with any word that begins with a "y":

```
[[ $ANSWER = y* ]]
```

[[...]] vs [...]

- [[has a '=~' regex match operator.
  - For example, the following will match a line (stored in the shell variable line) if there is a sequence of characters in the value consisting of any number, including zero, of space characters, zero or one instances of 'a', then a 'b':

```
[[ $line =~ [[:space:]]*(a)?b ]]
```

- [only supports a single condition; multiple tests with the '&&' and '||' operators must be in separate [brackets, while [[allows use of parentheses and the !, &&, and || logical operators within the]] brackets to combine subexpressions.
  - For example, with [[, you can test:

```
[[ (1 -le 2) && (2 -le 3) ]]
```

  whereas with [, you would do:

```
[ (1 -le 2) ] && [ (2 -le 3) ]
```

CIS 191

Linux and Unix Skills

LECTURE 7

Lecture Plan

Shell Scripting II

1. I/O
2. Compound Commands
3. Shell Functions
4. Some Common Errors and Debugging

I/O

read

read [-rs] [-a *aname*] [-d *delim*] [-n *nchars*] [-N *nchars*] [-p *prompt*] [-t *timeout*] [*name* ...]

- One line is read from the standard input, split into words, and the first word is assigned to the first *name*, the second word to the second *name*, and so on.
- If there are more words than names, the remaining words and their intervening delimiters are assigned to the last *name*.
- If there are fewer words read from the input stream than names, the remaining names are assigned empty values.
- The backslash character '\' is used to remove any special meaning for the next character read and for line continuation.
- If no names are supplied, the line read is assigned to the variable REPLY.

read

read [-rs] [-a *aname*] [-d *delim*] [-n *nchars*] [-N *nchars*] [-p *prompt*] [-t *timeout*] [*name* …]

- **-s** : Silent mode; if input is coming from a terminal, characters are not echoed.
- **-a** *aname* : The words are assigned to sequential indices of the array variable *aname*, starting at 0. All elements are removed from *aname* before the assignment. Other *name* arguments are ignored.
- **-d** *delim* : The first character of *delim* is used to terminate the input line, rather than newline.
- **-n** *nchars* : **read** returns after reading at most *nchars* characters.
- **-N** nchars : read returns after reading exactly nchars; delimiter characters encountered are not treated specially and do not cause read to return until nchars characters are read.
- **-p** prompt : Display prompt, without a trailing newline, before attempting to read any input.
- **-t** timeout : Cause read to time out and return failure if a complete line of input (or a specified number of characters) is not read within timeout seconds.

read

read [-rs] [-a *aname*] [-d *delim*] [-n *nchars*] [-N *nchars*] [-p *prompt*] [-t *timeout*] [*name* …]

- If the option **-r** is given, backslash does not act as an escape character, but is instead considered to be part of the line.
- **CAUTION**: You should almost always use the **-r** option with **read**.
- The characters in the value of the IFS variable are used to split the line into words.
- By default, IFS consists of whitespace characters, hence read ignores all leading and trailing whitespace characters.
- If you do not wish for all leading and trailing whitespace characters to be ignored, set the IFS variable to the empty string temporarily like so:

```
IFS="" read -r input
```

echo

echo [-neE] [*arg* …]

- Output the *args*, separated by spaces, terminated with a newline.
- **-n** : suppresses the trailing newline.
- **-e** : enables interpretation of backslash-escaped characters.
- **-E** : disables interpretation of backslash-escaped characters, even on systems where they are interpreted by default.

printf

printf [-v *var*] *format* [*arguments*]

- Write the formatted *arguments* to the standard output under the control of the *format*.
- -v *var* : assigns to the variable *var* rather than printing to standard output.
- A format specifier is of the form:

%[flags][width][.precision][length]specifier

where the *specifier* character at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument.

printf specifier	Output	Example
%c	Character	a
%s	String of characters	sample
%u	Unsigned decimal integer	7235
%d, %i	Signed decimal integer	-7235
%e, %E	Scientific notation	3.9265e+2, 3.9265E+2
%f, %F	Decimal floating-point	392.65
%g, %G	Use the shortest representation: %e or %f	392.65
%a, %A	Hexadecimal floating-point	-0xc.90fep-2, -0xc.90fEP-2
%x, %X	Unsigned hexadecimal integer	7fa, 7FA
%o	Unsigned octal	610
%%	Print a percent sign	%

Printf Examples

```
> printf "Characters: %c %c\n" "a" "b"
Characters: a b
> printf "%s \n" "A string"
A string
> printf "Preceding with blanks: %10d\n" "1977"
Preceding with blanks:       1977
> printf "Preceding with zeroes: %010d\n" "1977"
Preceding with zeroes: 0000001977
printf "Some different radices: %d %x %o %#x %#o \n" "65" "65" "65" "65" "65"
Some different radices: 65 41 101 0x41 0101
```

Compound Commands

if command

- The syntax of the **if** command is:

```
if test-commands; then
  consequent-commands;
[elif more-test-commands; then
  more-consequents;]
[else alternate-consequents;]
fi
```

- The *test-commands* list is executed, and if its return status is zero, the *consequent-commands* list is executed.

if command

- The syntax of the **if** command is:

```
if test-commands; then
  consequent-commands;
[elif more-test-commands; then
  more-consequents;]
[else alternate-consequents;]
fi
```

- If *test-commands* returns a non-zero status, each **elif** list is executed in turn, and if its exit status is zero, the corresponding *more-consequents* is executed and the command completes.

if command

- The syntax of the **if** command is:

```
if test-commands; then
  consequent-commands;
[elif more-test-commands; then
  more-consequents;]
[else alternate-consequents;]
fi
```

- If '*else alternate-consequents*' is present, and the final command in the final if or elif clause has a non-zero exit status, then *alternate-consequents* is executed.

if command

- The syntax of the **if** command is:

```
if test-commands; then
  consequent-commands;
[elif more-test-commands; then
  more-consequents;]
[else alternate-consequents;]
fi
```

- The return status is the exit status of the last command executed, or zero if no condition tested true.

Sorting 3 Numbers Using If Commands

```
if [ "$1" -le "$2" ] ; then
       if [ "$2" –le "$3" ] ; then first="$1" ; second="$2" ; third="$3"
       elif [ "$1" -le "$3" ] ; then first="$1" ; second="$3" ; third="$2"
       else first="$3" ; second="$1" ; third="$2" ; fi
elif [ "$2" -le "$3" ] ; then
       first="$2"
       if [ "$1" -le "$3" ] ; then  second="$1" ; third="$3"
       else second="$3" ; third="$1" ; fi
else first="$3" ; second="$2" ; third="$1" ; fi
echo "$first is less than $second which is less than $third"
```

case Command

- The syntax of the **case** command is:

```
case word in
  [ [(] pattern [| pattern]…) command-list ;;]…
esac
```

- **case** will selectively execute the *command-list* corresponding to the first pattern that matches *word*.
- There may be an arbitrary number of case clauses, usually terminated by a ';;', in which case no subsequent matches are attempted after the first pattern match.
- It's a common idiom to use '*' as the final pattern to define the default case, since that pattern will always match.

case Command Example

```
echo -n "Enter the name of an animal: "
read ANIMAL
echo -n "The $ANIMAL has "
case "$ANIMAL" in
  horse | dog | cat) echo -n "four";;
  man | kangaroo ) echo -n "two";;
  *) echo -n "an unknown number of";;
esac
echo " legs."
```

for Command

- The syntax of the **for** command is:

```
for name [ [in [words …] ] ; ] do commands;
done
```

- Expand *words* and execute commands once for each member in the resultant list, with *name* bound to the current member.
- If '*in words*' is not present, the for command executes the *commands* once for each positional parameter that is set, as if 'in "$@"' had been specified.
- The exit status is the exit status of the last command that executes.
- If there are no items in the expansion of words, no commands are executed, and the exit status is zero.

Cartesian Product using for Command

```
#!/usr/bin/bash

for element in "$@" ; do
        for element1 in "$@" ; do
                printf "(%s, %s)\n" "$element" "$element1"
        done
done
```

C-style for Command

- The syntax of the C-style **for** command is:

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

- First, the arithmetic expression *expr1* is evaluated.
- The arithmetic expression *expr2* is then evaluated repeatedly until its evaluates to zero.
- Each time *expr2* evaluates to a non-zero value, commands are executed and the arithmetic expression *expr3* is evaluated.
- If any expression is omitted, the whole command behaves as if it evaluates to 1.
- The return value is the exit status of the last command in commands that is executed, or false if any of the expressions is invalid.

Cartesian Product using C-Style For Command

```
#!/usr/bin/bash

for ((index=1 ; index<="$#" ; index++)) ; do
        for ((index1=1 ; index1<="$#" ; index1++)) ; do
                printf "(%s, %s)\n" "${!index}" "${!index1}"
        done
done
```

while Command

- The syntax of the **while** command is:

```
while test-commands; do consequent-commands;
                                        done
```

- Execute *consequent-commands* as long as *test-commands* has an exit status of zero.
- The return status is the exit status of the last command executed in consequent-commands, or zero if none was executed.

while Command Example

```
#!/usr/bin/bash
read -p "Input an animal : " ANIMAL

while [ "$ANIMAL" != "dog" ] ; do
        read -p "Wrong animal! Try again... " ANIMAL
done

echo "Yes! The right animal is $ANIMAL"
```

Grouping Commands

- Bash provides two ways to group a list of commands to be executed as a unit.
- Placing a list of commands between parentheses '()' causes a subshell environment to be created and each of the commands in list to be executed in that subshell.
  - Since the list is executed in a subshell, variable assignments do not remain in effect after the subshell completes.
- Placing a list of commands between curly braces '{}' causes the list to be executed in the current shell context; no subshell is created.
  - **CAUTION**: The semicolon (or newline) following the list is required.

Grouping Commands

```
> A=1
> echo "PID of this shell is $BASHPID"
PID of this shell is 12747
> (echo "PID of this shell is $BASHPID" ; A=2)
PID of this shell is 12982
> echo "$A"
1
> { echo "PID of this shell is $BASHPID" ; A=2 ; }
PID of this shell is 12747
> echo "$A"
> 2
```

Redirecting Compound Commands

- Compound commands can do redirections e.g.

```
#!/usr/bin/bash
{ read -p "Input an animal : " ANIMAL
        while [ "$ANIMAL" != "dog" ] ; do
                echo -en "Wrong animal! Try again...\n"
                read ANIMAL
        done ;
} < "$1"
echo "Yes! The right animal is $ANIMAL"
```

CAUTION!

- The following command will only print the first line of file1 to stdout:

```
while read –u input; do
        echo "$input"
        ssh eniac "echo $input"
done < "$file1"
```

- This is because the ssh command accepts data from stdin and hence consumes the rest of the lines in file1 (any command reading from stdin e.g. `cat > /dev/null` would behave similarly).
- To prevent this from happening, make the ssh command read from /dev/null:

```
ssh eniac "echo $input" < /dev/null
```

Shell Functions

Shell Functions

- A shell function *name* is declared using any of these syntax:

```
name () compound-command
function name [()] compound-command
```

- For maximum portability, use the first syntax.
- The reserved word **function** is optional, and if the **function** reserved word is supplied, the parentheses are optional.
- In the most common usage the curly braces that surround the body of the function must be separated from the body by blanks or newlines.
- When using the braces, the compound-command must be terminated by a semicolon, a '&', or a newline.

Shell Functions

- When a function is executed, the arguments to the function become the positional parameters during its execution.
- When a function completes, the values of the positional parameters and the special parameter '#' are restored to the values they had prior to the function's execution.
- Variables local to the function may be declared with the **local** command; these variables are visible only to the function and the commands it invokes.
- Functions may be recursive.

Fibonacci Function Example

```
#!/usr/bin/bash
function fib {
	if [ "$1" -eq 0 ] || [ "$1" -eq 1 ] ; then
		echo "$1"
	else
		term1="$(fib $(($1 - 1)))" ; term2="$(fib $(($1 - 2)))"
		echo "$((term1 + term2))"
	fi
}
echo "$(fib $1)"
```

return vs exit

- If the command **return** is executed in a function, the function completes and execution resumes with the next command after the function call.
- If a numeric argument is given to **return**, that is the function's exit status; otherwise the function's exit status is the exit status of the last command executed before the return.
- The exit status of a function definition is zero unless a syntax error occurs or a **readonly** function with the same name already exists.
- When executed, the exit status of a function is the exit status of the last command executed in the body.

return vs exit Example

```
> function foo { echo 25 ; return 64 ; }
> foo
25
> echo "$?"
64
```

Variable Scoping

- Variables local to the function may be declared with the ***local*** built-in.
- These variables are visible only to the function and the commands it invokes.
- The shell uses ***dynamic scoping*** to control a variable's visibility within functions.
- With dynamic scoping, visible variables and their values are a result of the sequence of function calls that caused execution to reach the current function.
- The value of a variable that a function sees depends on its value within its caller, if any, whether that caller is the "global" scope or another shell function.

Dynamic Scoping Example

```
> function f1  { local var="f1 local" ; f2;}
> function f2  { echo "In f2, var = $var"; }
> var="global"
> f1
In f2, var = f1 local
```

Some Common Errors and Debugging

Some Common Errors

- Assigning reserved words or characters to variable names e.g. case=1.
- Using a hyphen or other reserved characters in a variable name or function name e.g. var-1=23.
- Using whitespace inappropriately e.g. var1 = 23 or [$a -le 5].
- Failing to terminate a code block within curly brackets with a semicolon or newline after the final command e.g. { ls -l; echo "Done." }.

Some Common Errors

- Mixing up = and -eq in a test e.g. ["$a" = 273].
- Mixing up < and -lt in a test e.g. ["$number" < 5].
- Failing to double-quote parameter expansions e.g. $x instead of "$x".
- Using Bash-specific functionality such as [[in a Bourne shell script (#!/bin/sh); this may cause unexpected behaviour on a non-Linux machine.

Debugging

- Bash can be run in debug mode by running the script with the option **-x**.
- When Bash runs with the **-x** option on, it prints out every command it executes before executing it.
- Alternatively, you can debug only parts of the script by setting and unsetting the +x and -x variables in the script e.g.

```
set -x                  # activate debugging from here
COMMANDS_TO_DEBUG
set +x                  # stop debugging from here
```

Bash Resources

- If you have a question, go here to see if someone has asked it before:

  http://mywiki.wooledge.org/BashFAQ

- If you want to learn about some more common bash pitfalls, go here:

  http://mywiki.wooledge.org/BashPitfalls

- If you want to learn about some of the best bash scripting practices, go here:

  http://mywiki.wooledge.org/BashGuide/Practices

CIS 191

Linux and Unix Skills

LECTURE 8

Lecture Plan

Python Scripting I

1. Everything is an Object!
2. Booleans
3. Numbers
4. Container Types: Tuples, Lists, Ranges, Strings, Sets, Dictionaries
5. More on Classes

Python Overview

- Python was created by Guido van Rossum in 1991.
- Like most shell languages, Python is interpreted and dynamically typed.
- Python's design philosophy emphasizes code readability and ease of use.
- Most currently existing Python code is written in Python 2, though many projects are migrating their code to Python 3 as Python 2 will not be supported after 2020.
- Use Python 3!

Python vs Bash

- The main advantage of Bash and other shells is that they allow you to execute commands as you would on the terminal.
- However, Python:
  - has a nicer syntax.
  - supports rich types and data- structures.
  - has better error messages.
  - has better support for handling errors.
  - has many libraries.
- However, both Python and Bash are fussy over whitespace!

Everything is an Object!

Everything is an Object!

- Every expression in Python is an ***object***, and almost every expression has attributes and methods.
- This means that the type of each expression *e* in Python is a ***class*** *C*.
- A ***class*** describes a type by defining ***attributes*** and ***methods***.
- Attributes describe the state of an object, while methods are functions that may be invoked by an object to modify its state.
- If an object *e* has type *C* where *C* is a class, then *e* has each of the attributes and methods defined by *C*.
- The built-in function ***type***(*object*) returns the type of *object*.
- The built-in function ***dir***(*object*) returns a list of valid attributes for *object*.
- Some other things that are objects in Python include classes, functions, modules and packages!
- Python has a special null object called ***None*** that is returned by functions that don't explicitly evaluate to a value.

Class and Object Example

```
>>> class Animal:
...     def __init__(self, species, name):
...             self.name = name
...             self.species = species
...     def change_name(self, name):
...             self.name = name
>>> d = Animal('Dog', 'Fido')
>>> d.name
'Fido'
>>> d.change_name('Buddy')
>>> d.name
'Buddy'
```

Booleans

Booleans

- There are two constant objects that belong to the bool class: True and False.
- Any object can be tested for truth value, for use in an if or while condition, or as operand of a boolean operation ("and", "or", or "not").
- By default, an object is considered true unless its class defines either a __bool__ method that returns False or a __len__ method that returns zero, when called with the object.
- "or" and "and" always return one of their operands e.g. 0 or "yay" evaluates to "yay".
- "or" and "and" are short-circuit operators i.e. they only evaluate the second argument if necessary e.g. 1 or print(2) evaluates to 1 and does not print 2.

Booleans

- There are eight Boolean comparison operators in Python: <, <=, >, >=, ==, !=, "is" (object identity) and "is not" (negated object identity).
- The behaviour of the "is and "is not" operators cannot be customized.
- Objects of different types, except different numeric types, never compare equal.
- Non-identical instances of a class normally compare as non-equal unless the class defines the __eq__ method.
- Instances of a class cannot be ordered with respect to other instances of the same class, or other types of object, unless the class defines enough of the methods __lt__, __le__, __gt__, and __ge__.

Numeric Types

Numeric Types

- Python has three distinct numeric types: ***integers, floating point numbers, and complex numbers***.
- Integers have unlimited precision.
- When a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other (integer is narrower than floating point which is narrower than complex).

Common Numeric Methods

- ***x + y*** evaluates to the sum of x and y.
- ***x – y*** evaluates to the difference of x and y.
- ***x * y*** evaluates to the product of x and y.
- ***x / y*** evaluates to the quotient of x and y.
- ***x // y*** evaluates to the floored quotient of x and y.
- ***x % y*** evaluates to the remainder of x / y.
- ***x ** y*** and ***pow(x, y)*** both evaluate to x to the power y.
- ***-x*** evaluates to the negation of x.
- ***abs(x)*** evaluates to the absolute value or magnitude of x.
- ***int(x)*** evaluates to x converted to integer.
- ***float(x)*** evaluates to x converted to floating point.

Common Numeric Methods

- ***complex(re, im)*** evaluates to a complex number with real part re and imaginary part im.
- ***c.conjugate()*** evaluates to the conjugate of the complex number c.
- ***divmod(x, y)*** evaluates to the pair (x // y, x % y).
- ***round(x, n)*** evaluates to x rounded to n digits, rounding half to even if x is an int or a float.
- ***x | y*** evaluates to the bitwise or of x and y.
- ***x ^ y*** evaluates to the bitwise exclusive or of x and y.
- ***x & y*** evaluates to the bitwise and of x and y.
- ***x << n*** evaluates to x shifted left by n bits.
- ***x >> n*** evaluates to x shifted right by n bits.
- ***~ x*** evaluates to the bits of x inverted (i.e. –x – 1).

Numeric Method Examples

```
>>> 1+1
2
>>> 16.9 - 20.0
-3.1000000000000014 # urgh floating points!
>>> 3 * 4.0
12.0
>>> 1 / 2
0.5
>>> 1 // 2
0
>>> 6 % 2.1
1.7999999999999998 # urgh floating points again!
```

Numeric Method Examples

```
>>> -81
-81
>>> divmod(6, 2.1)
(2.0, 1.799999999999998)
>>> 2 ** 3
8
>>> int(54.5)
54
>>> float(54)
54.0
>>> round(2.145686649848, 5)
2.14569
```

Numeric Method Examples

```
>> complex(1.0, 2.0)
(1+2j)
>>> >>> complex(1.0, 2.0).conjugate()
(1-2j)
>>> complex(1.0, 2.0).conjugate()
(1-2j)
>>> 8 | 3 # 1010 | 0011 = 1011
11
>>> 11 & 13 # 1011 & 1101 = 1001
9
```

Numeric Method Examples

```
>>> 10 ^ 5 # 1010 ^ 0101 = 1111
15
>>> 16 << 2
64
>>> 16 >> 2
4
>>> ~32
-33
```

Container Types

Sequence Types

- There are three basic sequence types: ***tuples***, ***lists***, and ***ranges***.
- Tuples are immutable sequences that store items of variable type.
- Lists are mutable sequences of arbitrary length that store items of variable type.
- Ranges are immutable sequence of numbers that are commonly used for looping a specific number of times in for loops.
- The range type has an advantage over a regular list or tuple in that a range object will always take the same (small) amount of memory, no matter the size of the range it represents (as it only stores the start, stop and step values, calculating individual items and subranges as needed).
- Other commonly used sequence types include str and bytes, which are immutable sequences of Unicode code points and bytes respectively.

Tuples

- Tuples are immutable sequences that store items of variable type.
- Tuples may be constructed using parentheses, commas, or the tuple type constructor

```
>>> (1, 2, 3)
(1, 2, 3)
>>> 1, 2, 3
(1, 2, 3)
>>> tuple('abc')
('a', 'b', 'c')
```

Lists

- Lists are mutable sequences of arbitrary length that store items of variable type.
- Lists may be constructed using square brackets, the list type constructor, or list comprehensions:

```
>>> [1, 2, 3]
[1, 2, 3]
>>> list('abc')
['a', 'b', 'c']
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Ranges

- Ranges are immutable sequence of numbers that are commonly used for looping a specific number of times in for loops.
- Ranges are constructed using the range type constructor in three ways:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
```

Common Sequence Methods

- ***x in s*** evaluates to True if the object s is in the sequence s.
- ***x not in s*** evaluates to True if the object x is not in the sequence s.
- ***s + t*** evaluates to the concatenation of the sequences s and t.
- ***s * n*** and ***n * s*** with n an integer, s not a range object, evaluate to s + s + … + s with n occurrences of s.
- ***len(s)*** evaluates to the length of s.
- ***min(s)*** and ***max(s)*** evaluate to the smallest and largest item of s, assuming that all members of s can be compared to each other.
- ***s.index(x)*** evaluates to the first occurrence of x in s.
- ***s.index(x, i)*** evaluates to the first occurrence of x in s at or after index i.
- ***s.index(x, i, j)*** evaluates to the first occurrence of x in s at or after index i and before index j.
- ***s.count(x)*** evaluates to the total number of occurrences of x in s.

Sequence Methods Examples

```
>>> 1 in [1, "a", []]
True
>>> 2 not in (3,8)
True
>>> [1,2] + [3,4,5]
[1, 2, 3, 4, 5]
>>> ["a" ] * 10
['a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a', 'a']
>>> len(2 * [1,1,1])
6
```

Sequence Methods Examples

```
>>> min(["dog", "cat", "snake"]), max([1, 2, 3])
('cat', 3)
>>> ["a", 54, "b", "k", True, "b", (), "b", (), "b"].index("b")
2
>>> ["a", 54, "b", "k", True, "b", (), "b", (), "b"].index("b", 3)
5
>>> ["a", 54, "b", "k", True, "b", (), "b", (), "b"].index("b", 6, 9)


>>> ["a", 54, "b", "k", True, "b", (), "b", (), "b"].count("b", 6, 9)
4
```

Sequence Slicing

- ***s[i]*** evaluates to the item at index i of s.
- ***s[-i]*** evaluates to the item at index length(s) – i of s.
- ***s[i:j]*** evaluates to the subsequence of s starting from index i to index j-1.
- ***s[i:]*** evaluates to the subsequence of s starting from index i to index length(s) – 1.
- ***s[:i]*** evaluates to the subsequence of s starting from index 0 to index i – 1.
- ***s[:]*** evaluates to a shallow copy of s.
- ***s[i:-j]*** evaluates to the subsequence of s starting from index i to index length(s) – j – 1.
- ***s[i:j:k]*** evaluates to the subsequence of s containing the kth elements starting from index i to index j – 1.
- ***s[i:j:-k]*** evaluates to the subsequence of s containing the kth elements starting from index j to index i in reverse order.

Sequence Slicing Example

```
>>> s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[4]
4
>>> s[-2]
8
>>> s[3:7]
[3, 4, 5, 6]
>>> s[2:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> s[:7]
[0, 1, 2, 3, 4, 5, 6]
```

Sequence Slicing Example

```
>>> s[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[3:-4]
[3, 4, 5]
>>> s[2:7:3]
[2, 5]
>>> s[7:2:-2]
[7, 5, 3]
```

Common List Mutating Methods

- If k is any slice of s e.g. s[i], s[i:j], s[i:j:k] etc. then ***k = t*** sets the given slice of s to t.
- If k is any slice of s e.g. s[i], s[i:j], s[i:j:k] etc. then ***del k*** deletes the given slice from s.
- ***s.append(x)*** adds x to the end of s and evaluates to None.
- ***s.clear()*** removes all items from s.
- ***s.insert(i, x)*** inserts x into index i of s.
- ***s.pop([i])*** evaluates to the item at i (the last item if i is not given) and also removes it from s.
- ***s.remove(x)*** removes the first item from s where s[i] = x.
- ***s.reverse(x)*** reverses the items of s in place.
- ***s.sort()*** sorts s in place, using only < comparisons between items.
- **CAUTION**: Do not use s = append(x), or clear, insert, remove, reverse, or sort!

List Mutating Methods Examples

```
>>> s = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[3] = "b"
>>> s
[0, 1, 2, 'b', 4, 5, 6, 7, 8, 9]
>>> s[::2] = "vwxyz"
>>> s
['v', 1, 'w', 'b', 'x', 5, 'y', 7, 'z', 9]
>>> s.append("yay")
>>> s
['v', 1, 'w', 'b', 'x', 5, 'y', 7, 'z', 9, 'yay']
```

List Mutating Methods Examples

```
>>> s.clear()
>>> s
[]
>>> s.insert(0, 0); s.insert(1, 1); s.insert(1, 2)
[0, 2, 1]
>>> s.pop()
1
>>> s
[0, 2]
```

List Mutating Methods Examples

```
>>> s.append(2)
>>> s
[0, 2, 2]
>>> s.remove(2)
>>> s
[0, 2]
>>> s.reverse()
>>> s
[2, 0]
>>> s.sort()
>>> s
[0, 2]
```

Strings

- ***Strings*** are immutable sequences of Unicode code points.
- String literals are written in a variety of ways:
  - Single quotes: 'allows embedded "double" quotes'
  - Double quotes: "allows embedded 'single' quotes".
  - Triple quoted: '''Three single quotes''', """Three double quotes"""
- Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.
- Strings may also be created from other objects using the str constructor.
- Strings implement all of the common sequence Methods.

Common String Methods

- ***s.find(sub[, start[, end]])*** evaluates to the lowest index in s where substring sub is found within the slice s[start:end]; -1 if sub is not found.
- ***s.count(sub[, start[, end]])*** evaluates to the number of non-overlapping occurrences of substring sub in the slice s[start:end].
- ***s.endswith(suffix[, start[, end]])*** and ***s.startswith(suffix[, start[, end]])*** evaluate to True if s starts and ends with suffix in the slice s[start:end] respectively.
- ***s.join(iterable)*** evaluates to a string which is the concatenation of the strings in iterable, with s being the separator between elements.
- ***s.split(sep)*** evaluates to a list of the words in s using sep as the delimiter string.
- ***s.strip()*** evaluates to a copy of s with the leading and trailing characters removed.

String Method Examples

```
>>> "I want to search for this somewhere".find("this")
21
>>> "blah blah blah".count("blah")
3
>>> "I end with endswith".endswith("endswith")
True
>>> " ".join(["Let's", "make", "a", "sentence"])
"Let's make a sentence"
>>> "      I hate whitespace       ".strip()
'I hate whitespace'
>>> "smaina:x:44409:44409:Solomon Aduol Maina:/home1/s/smaina:/pkg/bin/bash".split(":")
['smaina', 'x', '44409', '44409', 'Solomon Aduol Maina', '/home1/s/smaina', '/pkg/bin/bash']
```

Formatting Strings

- ***s.format()*** allows you to perform a string formatting operation e.g.

```
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> import math
>>> '{:s} to five decimal digits is {:.5f}'.format('pi', math.pi)
'pi to five decimal digits is 3.14159'
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

Sets

- A set is an unordered collection of distinct ***hashable*** objects (objects with an __hash__ and __eq__ method).
- Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference and symmetric difference.
- Since sets are unordered, they do not support indexing or slicing.
- There are two built-in set types, ***set***, which is mutable, and ***frozenset*** which is immutable.
- Objects of type set and frozenset are created using their respective type constructors, and non-empty sets (not frozensets) can also be created by a placing comma-separated list of elements within braces, for example {'jack', 'sjoerd'}.

Common Set Methods

- ***len(s)*** evaluates to the number of elements in s.
- ***x in s*** and ***x not in s*** evaluate to True iff x is in s and x is not in s respectively.
- ***s.isdisjoint(t)*** evaluates to True iff s and t do not share any elements.
- ***s <= t*** evaluates to True iff all the elements in s are in t.
- ***s >= t*** evaluates to True iff all the elements in t are in s.
- ***s == t*** evaluates to True iff all the elements in s are in t and all the elements in t are in s.
- ***s < t*** evaluates to True iff all the elements in s are also in t, but no element of t is in s.
- ***s > t*** evaluates to True iff all the elements in t are also in s, but no element of s is in t.
- ***s | t*** evaluates to a set containing only elements that are in either of s or t.
- ***s & t*** evaluates to a set containing only elements that are in both s and t.
- ***s – t*** evaluates to a set containing only elements that are in s but not in t.
- ***s ^ t*** evaluates to a set containing only elements that are in either of s or t but not in both.

Set Method Examples

```
>>> len({1,1,2,2})
2
>>> 1 in {1,1}, 2 in {1,3}
(True, False)
>>> {1,2,3}.isdisjoint({4,5,6})
True
>>> {1} <= {1,2,3}
True
>>> {1,2,3} == {1,1,2,2,3,3}
True
>>> {1,2,3} < {1,1,2,2,3,3}
False
```

Set Method Examples

```
>>> {1,2,3} | {2,3,4}
{1, 2, 3, 4}
>>> {1,2,3} & {2,3,4}
{2, 3}
>>> {1,2,3} - {2,3,4}
{1}
>>> {1,2,3} ^ {2,3,4}
{1, 4}
```

Common Set Mutating Methods

- ***s |= others | …*** updates s, adding to it all elements from others.
- ***s &= others & …*** updates s, keeping only elements found in it and all others.
- ***s -= others - …*** updates s, removing elements found in others.
- ***s ^= other*** updates the set, keeping only elements found in either set, but not in both.
- ***s.add(x)*** adds x to s.
- ***s.remove(x)*** removes x from s. Raises a KeyError if x is not in s.
- ***s.discard(x)*** removes x from s. Does not raise a KeyError if x is not in s.
- ***s.pop()*** removes an arbitrary element from. Raises a KeyError if s is empty.
- ***s.clear()*** removes all elements from s.
- **CAUTION**: Do not use s = add(x), or, remove, discard, or clear!

Set Mutating Methods Examples

```
>>> s = {1,2,3}
>>> s |= {4,5} | {5,6}
>>> s
{1, 2, 3, 4, 5, 6}
>>> s &= {2,3,4,5} & {4,2}
>>> s
{2, 4}
>>> s -= {4,5,6}
>>> s
{2}
```

Set Mutating Methods Examples

```
>>> s ^= {2,3,4}
>>> s
{3, 4}
>>> s.add(1)
>>> s
{1, 3, 4}
>>> s.remove(3)
>>> s
{1, 4}
```

Set Mutating Methods Examples

```
>>> s.discard(4)
>>> s
{1}
>>> s.add(2)
>>> s
{1, 2}
>>> s.pop()
1
>>> s
{2}
```

Dictionaries

- ***Dictionaries*** are mutable objects that map hashable values to arbitrary objects.
- Dictionaries can be created by placing a comma-separated list of key : value pairs within braces, for example: {'jack': 4098, 'sjoerd': 4127}, by the dict constructor, or by using dictionary comprehensions:

```
>>> dict(one=1, two=2, three=3)
{'one': 1, 'two': 2, 'three': 3}
>>> { "a" : 1, () : 2, 3 : "c" }
{'a': 1, (): 2, 3: 'c'}
>>> { x : x + 1 for x in range(7, 11) }
{7: 8, 8: 9, 9: 10, 10: 11}
```

Common Dictionary Methods

- ***d[key]*** evaluates to item if the mapping (key, item) exists in d. Raises a KeyError if *key* is not in the map.
- ***d.get(key, default)*** evaluates to item if d contains the mapping (key, item), otherwise it evaluates to default.
- ***d[key]*** = value sets the mapping (key, value) to d.
- ***del d[key]*** removes the mapping (d, d[key]) from d. Raises a KeyError if key is not in the map.
- ***d.pop(key)*** removes the mapping (d, d[key]) and evaluates to d[key] if such a mapping exists. Raises a KeyError if no such mapping exists.
- ***d.pop(key, default)*** removes the mapping (d, d[key]) and evaluates to d[key] if such a mapping exists, otherwise evaluates to default.

Common Dictionary Methods

- ***key in d*** and ***key not in d*** evaluate to True and False if d has or does not have key *key* respectively.
- ***len(d)*** evaluates to the number of mappings in d.
- ***d.keys()*** evaluates to a new view of the keys in d.
- ***d.values()*** evaluates to a new view of the values in d.
- ***d.popitem()*** removes and evaluates to a (key, value) pair from the dictionary. Pairs are returned in Last In First Out (LIFO) order.
- ***d.update(other)*** updates the dictionary with the key/value pairs from other, overwriting existing keys.
- ***list(d)*** evaluates to a list of all the keys used in the dictionary d.

Dictionary Method Examples

```
>>> d={1:"a", "b":[1,2,3]}
>>> d["b"]
[1, 2, 3]
>>> d.get(2, "blah")
'blah'
>>> del d[1]
>>> d[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> d.pop("b")
[1, 2, 3]
```

Dictionary Method Examples

```
>>> d={1:2, 2:4, 3:4}
>>> 4 in d
False
>>> list(d.keys())
[1, 2, 3]
>>> list(d.values())
[2, 4, 4]
>>> d.popitem()
(3, 4)
>>> d.update({1:"a"})
>>> d[1]
'a'
>>> list(d)
[1, 2]
```

More on Classes

Duck Typing

- ***Duck typing*** in Python enables you to use the same attribute or method name for objects of different type; if it walks like a duck and it quacks like a duck, then it must be a duck.
- For example, you can use indexing notation with a custom class simply by implementing a __getitem__ method e.g.

```
>>> class Foo:
...     def __getitem__(self, n):
...         return ("yay " * (n-1)) + "yay"
...
>>> Foo()[4]
'yay yay yay yay'
```

Instance Attributes

- If a method f is defined using

  def f(arg1, arg2, …)

then any attribute defined in f using the notation "arg1.name" is only available to individual instances and not to all class instances; such attributes are called ***instance attributes***.

- Always use the name 'self' for the first argument to methods!
- When a class defines an __init__ method, class instantiation automatically invokes __init__ for the newly-created class instance.
- Initialize instance attributes in the __init__ method!
- Instance attributes can also be initiated in other methods.

Class Attributes

- A ***class attribute*** is any attribute defined inside a class but outside of a method.
- Class attributes are shared by all instances of a class.
- Both class and instance attributes are accessible through the notation "arg1.var" in methods with the first argument named "arg1" (usually arg1 is "self"), and an instance attribute hides a class attribute with the same name when accessed in this way.
- Class attributes can be used as defaults for instance attributes, but using mutable values there can lead to unexpected results.

Class Attributes vs Instance Attributes

```
class Animal:
        names = []                                  # Class attribute
        def __init__(self, name):
                self.my_names = [name]              # Instance attribute
        def add_name(self, names):
                self.names += names
>>> d = Animal('Fido')
>>> e = Animal('Buddy')
>>> d.add_name(d.my_names)                  # Adds Fido to e.names as well!
>>> print("e.names is " + e.names " and e.my_names is " + e.my_names)
e.names is ['Fido'] and e.my_names is ['Buddy']
```

Inheritance

- A class C may inherit attributes and methods defined in another class D; D is said to be the ***derived class*** and D the ***base class***.
- Derived attributes and methods override the same attributes and methods in the base class.
- If a derived class does not define an attribute or method but the base class does, then the attribute or method in the base class is used; this rule is applied recursively if the base class itself is derived from some other class.
- The in-built function ***super*** enables a derived class to use attributes and methods defined in the base class.
- The built-in attribute C.__bases__ evaluates to a tuple of base classes of a class object C.

Inheritance Example

```
>>> class Dog(Animal):
...     def __init__(self, name):
...             super().__init__(name)                  # execute Animal's __init__
...             for i in range(len(self.my_names)):
...                     self.my_names[i] = 'Dog ' + self.my_names[i]
>>> d = Dog('Fido')
>>> d.my_names
'Dog Fido'
```

Multiple Inheritance

- A class definition with multiple base classes is defined using the syntax:

```
class DerivedClassName(Base1, Base2, …, Base2):
```

- If an attribute is not found in DerivedClassName, it is searched for in Base1, then (recursively) in the base classes of Base1, and if it was not found there, it is searched for in Base2, and so on.
- The built-in attribute C.__mro__ evaluates to a tuple of classes that are considered when looking for base classes during method resolution.

CIS 191

Linux and Unix Skills

LECTURE 9

Lecture Plan

Python Scripting II

1. Modules and Packages
2. I/O
3. Statements: assignment, pass, raise, try, with, if, for, while
4. Functions and Lambda Expressions

Modules and Packages

Modules

- A ***module*** is a file containing Python definitions and statements.
- The module name is the file-name with the suffix .py removed.
- Definitions from a module can be imported into other modules or into the ***main*** module, the collection of variables that you have access to in a script executed at the top level.

Packages

- ***Packages*** are a way of structuring Python's module namespace by using "dotted module names" e.g. the module name A.B designates a submodule named B in a package named A.
- To create a package, create a directory with the name of the package, put the package modules and sub-packages in the directory, and create a file called __init__.py in each sub-package.
- The __init__.py files are required to make Python treat directories containing the file as packages.
- __init__.py can just be an empty file, but it can also execute initialization code for the package.

Package File System Layout Example

```
parent/
        __init__.py
        one/
                __init__.py
                file1.py
        two/
                __init__.py
                file2.py
                one/
                        __init__.py
                        file21.py
```

Importing Modules

- Modules can import other modules using the ***import*** statement e.g.

```
import foo
```

- Using the module name you can access the functions and variables defined in the module using the syntax "module_name.attribute_name" e.g.

```
foo.bar
```

- You can import specific functions and variables from a module directly using the syntax "from module_name import name_1, … , name_k" e.g.

```
from foo import bar1, bar2
```

- You can import all functions and variables defined in a module using the syntax "from module_name import *" e.g.

```
from foo import *
```

if __name__ == "__main__"

- A module's __name__ attribute is set equal to '__main__' when read from standard input, a script, or from an interactive prompt.
- If the following piece of code exists in a Python file "file1.py":

```
if __name__ == "__main__":
    SOME_PYTHON_CODE
```

then SOME_PYTHON_CODE will be executed if "file1.py" is executed as a script.

- In contrast, if "file1" is imported with the import keyword from a different file or interactive session, then SOME_PYTHON_CODE will not be executed.

Some Commonly Used Modules

- ***math*** module: provides access to the mathematical functions defined by the C standard (trig functions, constants such as pi and e, hyperbolic functions etc)
- ***sys*** module : provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter (sys.argv is the list of command line arguments passed to a Python script).
- ***os*** module : provides a portable way of using operating system dependent functionality.
- ***NumPy*** module: the fundamental package for scientific computing with Python, containing a powerful N-dimensional array object, sophisticated (broadcasting) functions, tools for integrating C/C++ and Fortran code, and useful linear algebra, Fourier transform, and random number capabilities.
- etc.

I/O

Opening Files

- The easiest way to create a text stream is with ***open()***, optionally specifying an encoding,

```
f = open("myfile.txt", "r", encoding="utf-8")
```

while the easiest way to create a binary stream is with open() with 'b' in the mode string:

```
f = open("myfile.jpg", "rb")
```

- You may specify an optional string that specifies the mode in which the file is opened.
- The default mode is 'r' which means open for reading in text mode, 'w' for writing (truncating the file if it already exists), 'r+' for both reading and writing, 'x' for exclusive creation, 'a' for appending, 't' for text (default), and 'b' for binary.
- The mode can only have one of create, read, write, or, append.

Closing Files

- Once you are done using a file object f, close the file using the ***close()*** method like so:

```
f.close()
```

- It is good practice to use the ***with*** keyword when dealing with file objects, the main advantage being that the file is properly closed after its suite finishes, even if an exception is raised at some point e.g.

```
>>> with open("myfile.txt", "r") as f:
...         read_data = f.read()
```

Reading from Files

- ***f.read()*** reads from f until EOF is encountered and returns the result as a single string.
  - If f is a binary stream, then a bytes object is returned.
- ***f.readline()*** reads and returns one line from the stream.
  - f.readline(n) reads at most n bytes from the line.
- ***f.readlines()*** reads and return a list of lines from the stream.
  - If called like so, f.readlines(n), no more lines will be read if the total size (in bytes/characters) of all lines so far exceeds n.
  - You can also iterate on file objects without calling file.readlines() using for loops:

```
for line in file: ...
```

Writing to Files

- ***f.write(s)*** writes the string s to f and returns the number of characters written.
  - If f is a binary stream, then write(b) writes the bytes-like object, b, and returns the number of bytes written.
- ***f.writelines(lines)*** write a list of lines to f.
- ***f.flush()*** flushes the write buffers of f if applicable.
  - f.flush() does nothing if no data has been added to the write buffers (e.g. using f.write()).
- ***f.close()*** flushes and closes f.
  - Once the file is closed, any operation on the file (e.g. reading or writing) will raise a ValueError.

File I/O Example

```
> echo -e "line 1\nline 2\nline3" > new && python3
>>> f = open("new", "a")
>>> f.write("line 4\n")
7
>>> f.writelines(["line 5\n", "\n", "line 7\n"])
>>> f.close()
>>> f = open("new", "r")
>>> f.readlines()
['line1\n', 'line2\n', 'line3\n', 'line 4\n', 'line 5\n', '\n', 'line 7\n']
>>> f.close()
```

I/O with Standard Streams

- ***sys.stdin***, ***sys.stdout***, and ***sys.stderr*** are file objects used by the interpreter for standard input, output, and errors.
- These streams are regular text files like those returned by the open() function.
- ***input***([prompt]) reads a line from stdin, converts it to a string (stripping a trailing newline) and returns that.
  - If the prompt argument is present, it is written to standard output without a trailing newline.
- ***print***(**objects, sep=' ', end='\n', file=sys.stdout*) prints objects to the text stream *file*, separated by sep and followed by *end*.

Standard Streams I/O Examples

```
#!/usr/bin/python3
import sys
if __name__ == "__main__":
	if sys.stdin.isatty():    # True if the stream is interactive
		response = input("Type something... ")
		print("You typed", response)
	else:
		print("These are the lines from stdin:")
		for line in sys.stdin:
			sys.stdout.write(line)
```

Parsing Command-Line Arguments

- The ***argparse*** module makes it easy to write user-friendly command-line interfaces.
- The following code is a Python program that takes a list of integers and produces either the sum or the max:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')

parser.add_argument('integers', metavar='N', type=int, nargs='+',
help='an integer for the accumulator')

parser.add_argument('--sum', dest='accumulate', action='store_const',
const=sum, default=max, help='sum the integers (default: find the max)')

args = parser.parse_args()

print(args.accumulate(args.integers))
```

Argparse Output Example

```
> ./prog.py
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: the following arguments are required: N
> ./prog.py 1 2 3
3
> ./prog.py 1 2 3 --sum
6
> ./prog.py 1 2 --sum 3
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: unrecognized arguments: 3
```

Argparse Help Message Example

```
> ./prog.py -h
usage: prog.py [-h] [--sum] N [N ...]
Process some integers.
positional arguments:
  N           an integer for the accumulator
optional arguments:
  -h, --help  show this help message and exit
  --sum       sum the integers (default: find the max)
```

add_argument and parse_args

- The most important method in the ArgumentParser class is the ***add_argument()*** method which tells the ArgumentParser object how to take the strings on the command line and turn them into objects.
- The parsed command-line arguments are then packaged as attributes of a single object using the ***parse_args()*** method.
- The first arguments passed to add_argument() must be either a series of flags, or a simple argument name.
- When parse_args() is called, optional arguments will be identified by the '-' prefix, and the remaining arguments will be assumed to be positional.

add_argument Parameters

- The ***metavar*** keyword argument specifies the name of the argument displayed in help messages.
- The ***help*** keyword argument is a string containing a brief description of the argument.
- The ***dest*** keyword argument determines the name of the attribute to be added to the object returned by parse_args().
- The ***default*** keyword argument, whose value defaults to None, specifies what value should be used if the command-line argument is not present.
- The ***type*** keyword argument specifies the type into which the parsed string should be converted.
  - type can take any function that takes a single string argument and returns the converted value.
- The ***required*** keyword argument specifies whether the argument is required.
- The ***choices*** keyword argument specifies a container object such as a list or set that restricts the values the argument can attain.

add_argument Parameters

- The ***action*** keyword argument specifies how the command-line arguments should be handled.
  - The default action is 'store' which just stores the argument value.
  - There are many other actions available including 'store_const', 'store_true', 'store_false', 'append' etc.
- The ***const*** keyword argument is used to hold constant values that are not read from the command line but are required for the various ArgumentParser actions.
  - With the 'store_const' and 'append_const' actions, the const keyword argument must be given. For other actions, it defaults to None.
- The ***nargs*** keyword argument associates a different number of command-line arguments with a single action.
  - N (an integer) : N arguments from the command line will be gathered together into a list.
  - '*' : All command-line arguments present are gathered into a list.
  - '+' : Like '*', but an error message will be generated if there wasn’t at least one command-line argument present.
  - '?' : One argument will be consumed from the command line if possible, and produced as a single item.
  - argparse.REMAINDER : All the remaining command-line arguments are gathered into a list.

Statements

Assignment and Pass Statements

- An ***assignment*** statement sets a named variable to an object e.g.

```
x = 1
```

- The ***pass*** statement does nothing when executed.
- The pass statement is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass     # a function that does nothing
class C: pass        # a class with no methods
```

Raise Statements

"raise" [expression]

- ***raise*** re-raises the last exception that was active in the current scope, or, if no such exists, evaluates expression as the exception object.
- The exception object must be either a subclass or an instance of BaseException e.g.

```
def divide(n, d):
        if not isinstance(n, complex) or not isinstance(d, complex):
                raise ValueError("One of the arguments is not a number")
        elif d == 0:
                raise ZeroDivisionError("The denominator is zero")
        else : return n/d
```

Try Statements

```
"try" ":" suite
      ("except" [expression ["as" identifier]] ":" suite)+
      ["else" ":" suite]
      ["finally" ":" suite]
```

- When an exception occurs in the ***try*** suite, the ***except*** clauses are inspected in turn until one is found that matches the exception.
- The exception is assigned to the target specified after the as keyword in that except clause, if present, and the except clause's suite is executed.
- When the end of this block is reached, execution continues normally after the entire try statement.
- The optional ***else*** clause is executed if the control flow leaves the try suite or no exception was raised.

Try Statements

```
"try" ":" suite
      ("except" [expression ["as" identifier]] ":" suite)+
      ["else" ":" suite]
      ["finally" ":" suite]
```

- The ***finally*** clause, if present, is always executed.
- If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved, the finally clause is executed, and the saved exception is re-raised at the end of the finally clause.
- If the finally clause itself raises an exception, then the new exception is raised.
- If the finally clause executes a return, break or continue statement, the saved exception is discarded.

Try Statement Example

```
import sys
try:
    if sys.argv[1] == '0': raise FileNotFoundError
    elif sys.argv[1] == '1': raise ZeroDivisionError
    elif sys.argv[1] == '2': pass
    else: raise ValueError
except FileNotFoundError: print("You broke Unix")
except ZeroDivisionError: print("You broke math")
else: print("You broke nothing")
finally: print("I'm always printed")
```

With Statements

"with" expression ["as" target] ("," expression ["as" target])* ":" suite

- The ***with*** statement is used to wrap the execution of a block with methods defined by a ***context manager*** i.e. an object with an __enter__() and __exit__() method.
- with statements allow common try…except…finally usage patterns to be encapsulated for convenient reuse.
- In a nutshell:
  1. The context expression is evaluated to obtain a context manager.
  2. The context manager's __enter__() is loaded for later use.
  3. The context manager's __exit__() is loaded for later use.
  4. The context manager's __enter__() method is invoked.
  5. The suite is executed.
  6. The context manager's __exit__() method is invoked.

With Statement Examples

- with statements are recommended when dealing with files e.g.

```
>>> with open("myfile.txt", "r") as f:
...         read_data = f.read()
```

- with statements are often also used when using synchronization objects e.g. in the following example, the lock is acquired before the block is executed and always released once the block is complete.

```
>>> import threading
>>> lock = threading.Lock()
>>> with lock:
...         # Critical section of code
...         ...
```

If Statements

```
"if" expression ":" suite
     ("elif" expression ":" suite)*
     ["else" ":" suite]
```

- An ***if*** statement selects exactly one of the suites by evaluating the expressions one by one until one is found to be true, then that suite is executed; no other part of the if statement is executed or evaluated.
- If all expressions are false, the suite of the else clause, if present, is executed.
- Python also supports ***conditional expressions*** that uses a similar syntax e.g. the following expression evaluates to "yay"

```
12 if 1 > 2 else "yay"
```

If Statement Example

```
import random
x = random.randint(-1, +1)
if not x:
     print("x is 0")
elif x < 0:
     print("x is -1")
else:
     print("x is +1")
```

For Statement

```
"for" target_list "in" expression_list ":" suite
      ["else" ":" suite]
```

- The expression list is evaluated once; it should yield an iterable object.
- An ***iterable*** object is an object that implements __iter__, which is expected to return an ***iterator*** object.
- An ***iterator*** is an object that implements __iter__ and __next__, which is expected to return the next element of the iterable object that returned it, and raise a StopIteration exception when no more elements are available.
- The suite is then executed once for each item provided by the iterator, in the order returned by the iterator.
- When the items are exhausted, the suite in the else clause, if present, is executed, and the loop terminates.

For Statement Example

```
>>> import random
>>> class WeirdIterator:
…     def __iter__(self) : return self
…     def __next__(self) :
…           if random.randint(0, 1): return "So random"
…           else: raise StopIteration
>>> for x in WeirdIterator() : print(x)
...
So random
```

While Statement

```
"while" expression ":" suite
    ["else" ":" suite]
```

- This repeatedly tests the expression and, if it is true, executes the first suite.
- If the expression is false, which may be the first time it is tested, the suite of the else clause, if present, is executed and the loop terminates.

While Statement Example

```
f = open('my_text_file.txt')
line = f.readline()
while line:
    print(line.strip())
    line = f.readline()
f.close()
```

Functions

Functions

```
"def" function_name "(" parameter_list ")" ":"
     suite
```

- A ***function*** definition defines a user-defined function object e.g.

```
def whats_on_the_telly(penguin=None):
     if penguin is None:
          penguin = []
     penguin.append("property of the zoo")
     return penguin
```

Default Parameter Values

- When one or more parameters in a function's parameter list have the form parameter=expression, the function is said to have ***default parameter values***.
- For a parameter with a default value, the corresponding argument may be omitted from a call, in which case the parameter’s default value is substituted.
- If a parameter has a default value, all following parameters up until a "*identifier" or "**identifier" parameter must also have a default value.
- Parameters without default values must appear before parameters with default values in a function's parameter list.

Default Parameter Values Example

```
>>> def f(arg1, arg2, arg3=25, arg4=64):
...   return (arg3, arg4)
...
>>> f(1,2)
(25, 64)
>>> f(1,2,3)
(3, 64)
>>> f(1,2,3,4)
(3, 4)
```

CAUTION!

- Default parameter values are calculated once, when the function is defined.
- A mutable object such as a list or dictionary used as a default value will be shared by all calls that don't specify an argument value for the corresponding slot e.g.

```
>>> def f(arg1=[]):
...     arg1.append(1)
...     return arg1
...
>>> f(), f(), f()
[1], [1, 1], [1, 1, 1]
```

Positional and Keyword Arguments

- If the form "*args" is present in a function's parameter list, it is initialized to a tuple receiving any excess ***positional*** (i.e. unnamed) ***arguments***, defaulting to the empty tuple.
- If the form "**kwags" is present in a function's parameter list, it is initialized to a new dictionary receiving any excess ***keyword*** (i.e. named) ***arguments***, defaulting to a new empty dictionary of the same type.
- Only one parameter of each of the forms "*args" and "**kwargs" may be defined, and no parameters may be defined after "**kwargs".
- Parameters after "*identifier" in a function's parameter list are keyword-only parameters and may only be passed as keyword arguments in function calls.
- Positional arguments must appear before keyword arguments in function calls.

Positional and Keyword Argument Examples

```
>>> def f(arg1, *args, arg2=0, **kwargs):
...     return (arg1, args, arg2, kwargs)
...
>>> f(1)
(1, (), 0, {})
>>> f(1,2,3,4,arg2=5,foo=6,bar=7)
(1, (2, 3, 4), 5, {'foo': 6, 'bar': 7})
>>> f(arg2=1, 2)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

Argument Binding in Function Calls

- When a function is called, keyword (i.e. named) arguments are bound to the corresponding formal keyword parameter in the function's parameter list.
- Positional (i.e. unnamed) arguments are bound to formal positional parameters one by one in the order given by the function's parameter list.
- When all arguments have been processed, the formal parameters that have not been assigned a value are bound to the corresponding default value from the function's parameter list.

Lambda Expressions

```
"lambda" [parameter_list] ":" expression
```

- ***Lambda expressions*** are used to create anonymous (i.e. unnamed) functions.
- The unnamed function object behaves like a function object defined with:

```
def <lambda>(parameters):
    return expression
```

- Functions created with lambda expressions cannot contain statements or annotations.

Lambda Expression Examples

```
>>> list(filter(lambda x: x > 27, [12, 4, 68, 88, 23, 100, 112]))
[68, 88, 100, 112]
>>> list(map(lambda x: x + 16, [12, 4, 68, 88, 23, 100, 112]))
[28, 20, 84, 104, 39, 116, 128]
>>> lambda x: for x in [1,2,3,4] : print(x)     # no statements allowed
  File "<stdin>", line 1
    lambda x: for x in [1,2,3,4] : print(x)
                  ^
SyntaxError: invalid syntax
```

CIS 191

Linux and Unix Skills

LECTURE 10

Lecture Plan

Version Control with Git

1. Overview
2. Basic Git Workflow
3. Branching and Rebasing
4. Undoing Changes
5. Git Some More!

Version Control

Version Control

- A ***Version Control System*** (VCS) enables you to record changes to a set of files over time so that you can recall specific versions later.
- VCS allows you to:
  1. revert selected files or an entire project back to a previous state,
  2. compare changes to a project over time, and
  3. see who last modified something that might be causing a problem, who introduced an issue and when
- Because of these benefits, VCS is a must when developing software, especially when developing software collaboratively.
- Many version control systems exist e.g. Git, Mercurial, Darcs, etc.

Git

- ***Git*** is a version control system created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development.
- Git is free and open-source software distributed under the terms of the GNU General Public License version 2.
- Git is a distributed VCS, giving each developer a local copy of the full development history, and changes are copied from one such repository to another.
- There exist many services that host projects developed using Git e.g. GitHub, GitLab, BitBucket etc.

Basic Git Workflow

git clone, git init

- You typically obtain a Git repository in one of two ways:
  1. by taking a local directory that is currently not under version control and turning it into a Git repository, using the ***git init*** command, or
  2. by cloning an existing Git repository from elsewhere using the ***git clone*** command.
- Both git clone and git init create a new subdirectory named .git that contains all of your necessary repository files.
- To turn the current directory into a Git repository, simply execute the command inside the directory:

```
git init
```

git clone

- To clone an existing Git repository over the internet, say the linux repo, execute a git clone command via https or ssh with the following commands respectively

```
git clone https://github.com/torvalds/linux.git
```

```
git clone git@github.com:torvalds/linux.git
```

- In order to clone via ssh, you must add your public key to your hosting site e.g. GitHub, BitBucket etc.

- You can also clone a Git repository from your own file-system e.g. to clone a Git repository found at the location ~/Documents/cool_git_project into the current directory, execute the command:

```
git clone ~/Documents/cool_git_project
```

git remote

- ***Remote repositories*** are versions of your project that are hosted somewhere else e.g. on the Internet or at a different location in your file-system.
- Use ***git remote*** to list the remote connections you have to other repos.
- **-v**: includes the URL of each connection.
- Use ***git remote show <name>*** to display some more info about the connection <name>.
- The commands

```
git remote add <name> <url>

git remote rm <name>

git remote rename <old-name> <new-name>
```

respectively add a new connection named <name> to the repository at <url>, remove the connection named <name>, and rename the connection named <old-name> to <new-name>.

Basic Git Workflow

- Once a Git repository has been established, the basic Git workflow goes something like this:
  1. You ***pull*** changes from the remote repository into your local repository.
  2. You ***modify*** files in your ***working tree***.
  3. You selectively ***add*** just those changes you want to be part of your next commit to the ***staging area***.
  4. You do a ***commit***, which takes the files as they are in the staging area and stores that snapshot permanently to your repository.
  5. You ***push*** changes from your local repository into the remote repository.

git fetch, git merge, git pull

- ***git fetch <remote>*** downloads a remote repository but does not integrate into the local repo.
- ***git merge*** integrates a remote repository with the local repo.
- Git may be able to integrate the local and remote repos without any conflicts.
- A merge will fail if the local and remote copies of a file differ in the same position/s.
- In case a merge fails, Git adds standard conflict-resolution markers to the files that have conflicts so that you can open them manually and resolve those conflicts.
- The command ***git pull <remote>*** combines a git fetch <remote> and a git merge in one command.

Merge Conflicts

- A section in a file with a merge conflict looks like this:

```
<<<<<<< HEAD
The quick brown fox jumps over the really lazy dog.
=======
The quick brown fox jumps over the laaaaaazy dog.
>>>>>>> commit_id
```

- The local version is displayed above the equals signs ('=======') while the remote version is displayed below the equals signs.
- In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself e.g. you can replace the conflict section with the merged version:

```
The quick brown fox jumps over the really laaaaaazy dog.
```

git status and git add

- Use ***git status*** to see the files that are not being tracked by git, the files that have been modified in the working tree but have not been added to the staging area, or the files in the staging area that do not appear in the latest commit.
- After inspecting the state of your repository with git status, use ***git add <files>*** to add <files> from your working tree to the staging area.
- This marks <files> as being ready to be committed.

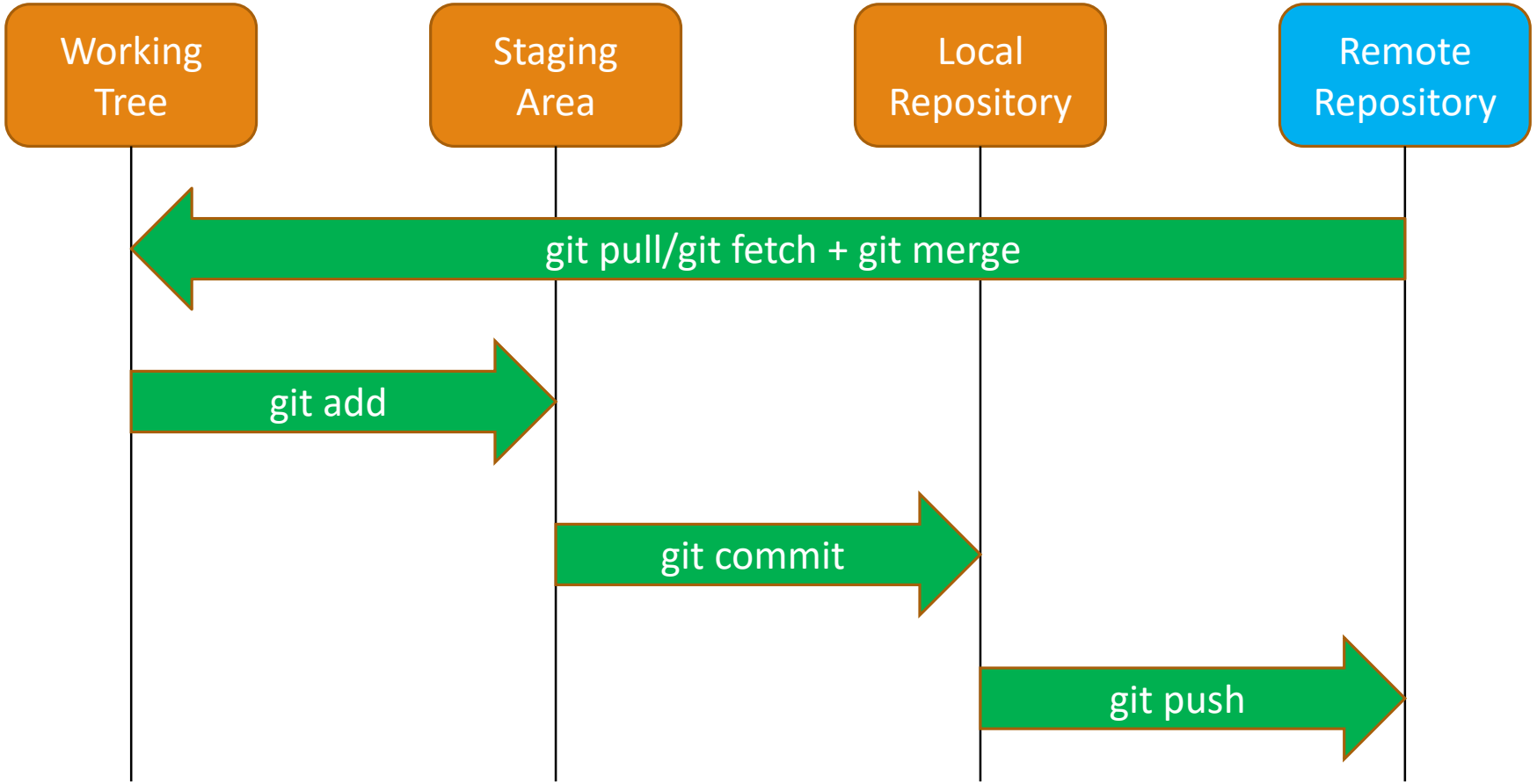
git commit

- Use ***git commit*** to move the current contents of the staging area permanently into your local repository.
- Executing git commit launches an editor that allows you to input a commit message.
- Use the **–m** option to give the commit message on the terminal instead.
- Use the **–a** option to automatically add every file that git is already tracking to the staging area; this lets you skip the git add part.
- Use informative commit messages!
- Commit often!

git push

- Use ***git push <remote>*** to upload your commits to <remote>.
- If you and someone else had previously pulled the same old commit from the remote repository but they push their new commits before you do, your push will be rejected.
- Git will ask you to pull their commits first and merge them into your local repository before you can push your new commits into the remote repository.

Basic Git Workflow



Branching and Rebasing

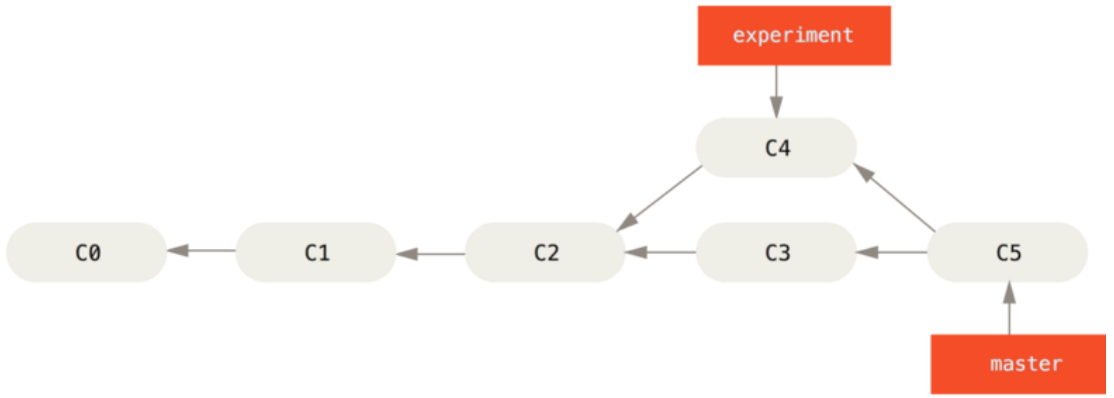
git branch, git checkout, git merge

- ***Branching*** enables you to diverge from the main line of development and continue to do work without messing with the main line.
- ***git branch*** lists existing branches, and highlights the current branch, marking it with an asterisk.
- Use ***git branch <branch>*** to create a new branch called <branch> from the current branch.
  - **-d**: delete the branch <branch>.
- Use ***git checkout <branch>*** to switch to the branch named <branch>.
  - **-b**: create a new branch named branch and switch to it.
- Use ***git merge <branch>*** to merge the branch <branch> into the currently checked out branch.
- If both branches have diverged, then git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.
- A merge may fail if the copies of a file in the two branches differ

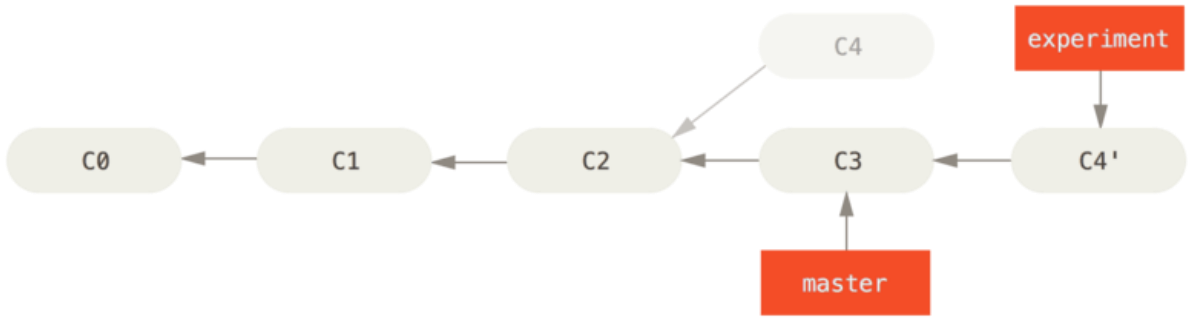
git rebase

- ***Rebasing*** is the process of moving or combining a sequence of commits to a new base commit.
- The command ***git rebase <old_branch>***
  1. saves all the changes made in the currently checked out branch from the common ancestor of <old_branch> and the current branch,
  2. resets the current branch to the same commit as <old_branch>, and
  3. applies the saved changes on the current branch in a new commit
- Rebasing hence leads to a linear project history.
- When rebasing leads to merge conflicts, fix the merge conflicts, git add the files with merge conflicts, and then use ***git rebase --continue*** to complete the rebasing.

Branching vs Rebasing



Branching



Rebasing

Branching vs Rebasing

- Branching and rebasing lead to the same files, but rebasing leads to a linear and therefore simpler history.
- On the other hand, if you rebase commits that other people have based their work on, then your collaborators will have to re-merge their work, after which you will have to re-merge your work…
- Only rebase local changes you've made but haven't shared; never rebase anything you've pushed somewhere.

HEAD

- Git maintains a pointer called ***HEAD*** that points to the most recent commit of the current branch.
- HEAD^ points to the parent of HEAD.
- HEAD^n points to the nth parent of HEAD (commits can have two parents in case of a merge).
- HEAD~ points to the parent of head (same as HEAD^).
- HEAD~n traverses the first parents the number of times you specify e.g. HEAD~2 points to the first parent of the first parent of HEAD.

Detached HEAD

- ***git checkout <commit>*** detaches HEAD and moves it to the specified commit, and updates the staging area and the working tree to the tree of <commit> while keeping local modifications to the files in the working tree.
- When you checkout a commit, HEAD becomes in a ***detached*** state, which means that HEAD refers to a specific commit, as opposed to referring to a named branch.
- **CAUTION**: Always create a new branch to make modifications to your files when HEAD is in a detached state, otherwise you will lose your modifications when you inevitably switch to an actual branch.
- Use git checkout <commit> followed by git checkout –b <branch> to create a new branch named <branch> based at <commit>.

Undoing Changes

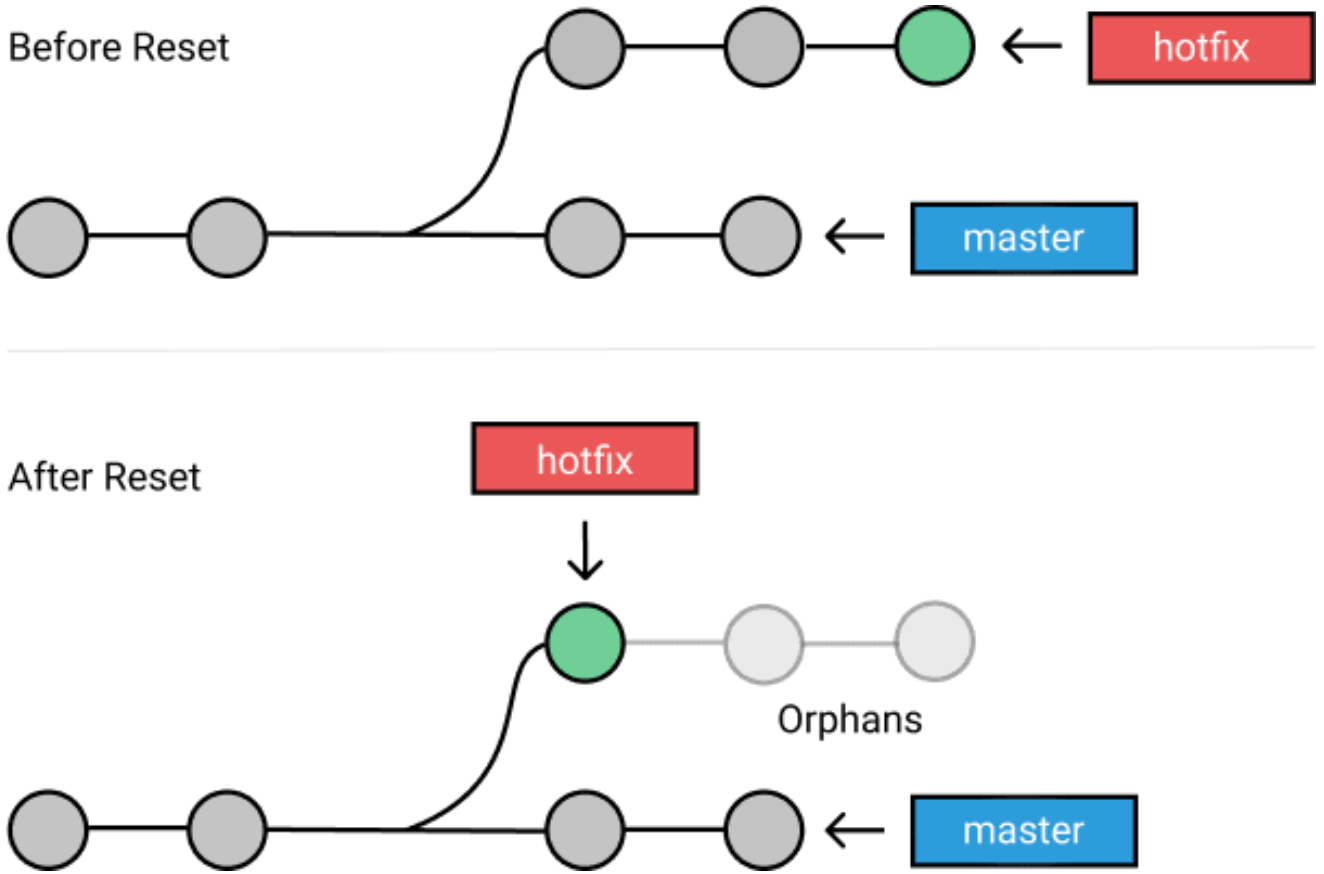
git log and git reflog

- Use ***git log*** to list the commits made in your repository in reverse chronological order starting from HEAD.
  - **--graph**: display an ASCII graph of the branch and merge history beside the log output.
  - **--abbrev-commit**: show only the first few characters of the SHA-1 checksum instead of all 40.
  - **--pretty=format**: show commits in an alternate format. Options include oneline, short, full, and fuller.
  - **--stat**: show statistics for files modified in each commit.
- ***git reflog*** is a log of where your HEAD and branch references have been for the last few months.
- HEAD@{n} refers to the nth prior value if HEAD in the reflog.
- The reflog is purely local; it is not included in pushes, fetches or clones.
- git reflog enables you to undo almost any changes within reason!

git reset

- ***git reset <files>*** unstages files without changing the working tree i.e. git reset <files> is the opposite of git add <files>.
- ***git reset [<mode>] [<commit>]*** resets the current branch head to <commit> and possibly updates the staging area, resetting it to the tree of <commit>, and the working tree depending on <mode>.
  - **--soft** : Does not touch the staging area or the working tree at all (but resets the head to <commit>, just like all modes do).
  - **--mixed** (the default) : Resets the staging area but not the working tree i.e., the changed files are preserved but not marked for commit.
  - **--hard**: Resets the staging area and working tree.

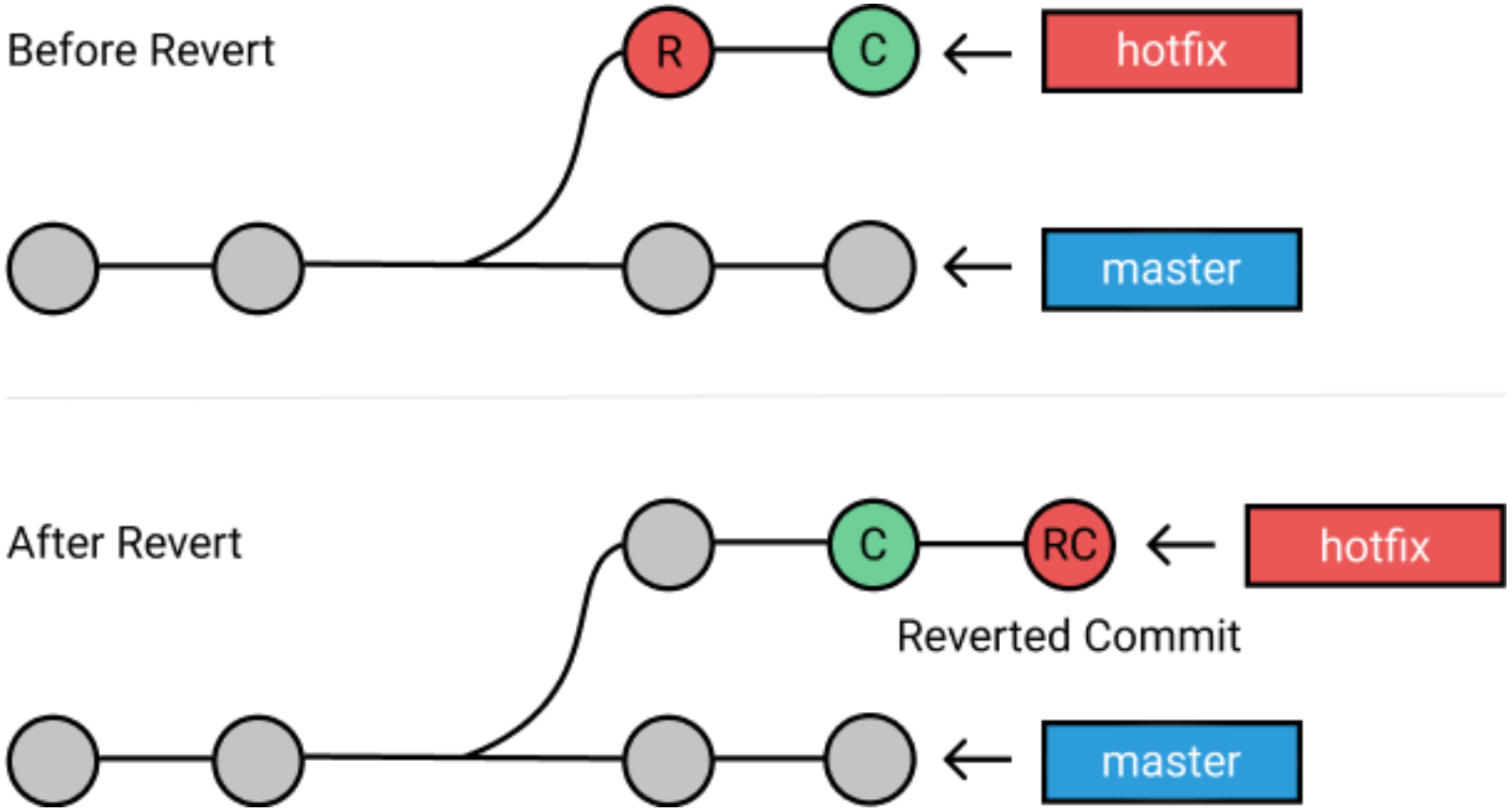
git reset


Before Reset
hotfix
master
After Reset
hotfix
Orphans
master

git revert

- ***git revert <commit>*** reverses the changes introduced by <commit>, and appends a new commit with the resulting inverse content.
- **CAUTION**: git revert <commit> DOES NOT reverse commits that occur after <commit> e.g. if commit1 leaves a file with a single line containing "1", commit2 leaves the lines "1", "2", and commit3 leaves the lines "1", "2" "3", then git revert commit2 will leave the lines "1", "3".
- git revert <commit> will cause merge a conflict if a subsequent commit changes a line that is also changed by <commit>.
- If a merge conflict occurs, fix the merge conflicts, git add the files with merge conflicts, and then use ***git revert --continue*** to complete the revert.

git revert


Before Revert
R
C
hotfix
master
After Revert
C
RC
hotfix
Reverted Commit
master

git checkout

- Use ***git checkout <file>*** to update <file> in the working tree to the version in the staging area.
- Use ***git checkout <commit> <file>*** to update <file> in the working tree to the version from <commit> e.g. to change "file.txt" to the version from two commits ago in the branch master, use the command:

```
git checkout master~2 file.txt
```

git stash

- Use ***git stash*** when you want to record the current state of the working directory and the staging area, but want to go back to a clean working directory.
- ***stash@{n}*** refers to the nth item on the stash list.
- ***git stash list*** lists the stash entries you currently have.
- Use ***git stash apply stash@{n}*** to go back to the working directory stored in stash@{n} (does not remove stash@{n} from the stash list).
  - **CAUTION**: Will fail if current working tree conflicts with applied stash!
- Use ***git stash show –p stash@{n}*** to show the changes recorded in the stash entry as a diff between the stashed contents and the commit back when the stash entry was first created.
- Use ***git stash clear*** to remove all the stash entries.

Summary

- Use git reset <file> to unstage <file>.
- Use git reset <commit> to switch the current branch to <commit>.
- Use git revert <commit> to undo the changes introduced by <commit>.
  - **CAUTION**: Will lead to merge conflicts if subsequent commits clash with <commit>!
- Use git checkout <file> to update <file> in the working tree to the version in the staging area.
- Use git checkout <file> <commit> to update <file> to the version in <commit>.
- Use git stash to clean the working directory.
- Use git stash apply stash@{n} to go back to the working directory stored in stash@{n}.
  - **CAUTION**: Will fail if current working tree conflicts with stash@{n}!

Git Some More!

git diff, git apply

- Use ***git diff*** to view the changes you made to the working tree relative to the staging area.
- Use ***git diff <commit>*** to view the changes you have in your working tree relative to the named <commit>.
- Use ***git diff --no-index <path> <path>*** to compare the given two paths on the filesystem.
- Use ***git apply <patch>*** to apply <patch> to the staging area (patch is normally the output of a git diff command).

git config

- ***git config*** enables you to specify Git configuration settings.
  - **--local**: uses repository config file (.git/config)
  - **--global**: uses user config file (~/.gitconfig)
  - **--system**: uses system-wide config file (/etc/config)
- git config allows you to configure many settings e.g.

```
> git config user.name "John Doe"                    # set user name
> git config user.email johndoe@example.com          # set email addr
> git config core.editor emacs                  # used e.g. for committing
> git config commit.template file.txt           # default commit message
> git config color.ui false               # turn of coloured terminal output
> git config help.autocorrect 5    # autocorrects after 5 tenths of a second
```

.gitignore

- A ***.gitignore*** file specifies intentionally untracked files that Git should ignore using syntax similar to wildcard patterns.
- Files specified in the .gitignore files are often compilation files e.g. object and assembly files.
- Each line in a .gitignore file specifies a pattern for files to be tracked or not tracked e.g.

```
> cat .gitignore
*.pdf            # do not track pdf files
/lib/            # do not track the lib/ directory
!/lib/must       # do not track lib/ except for /lib/must
```

git submodule

- A ***submodule*** is a repository embedded inside another repository.
- Submodules can be used for at least two different use cases:
  1. Using another project while maintaining independent history.
  2. Splitting a (logically single) project into multiple repositories and tying them back together.
- Use ***git submodule add <url>*** to add a submodule at the location <url>.
- A submodule and its super-project have separate commit histories.
- Use git config to enable regular commands such as pulls to recurse into submodules with the command:

```
git config submodule.recurse true
```

Hooks

- ***git hooks*** are scripts that run automatically every time a particular event occurs in a git repository.
- Common uses for git hooks include running tests on new code, updating dependencies etc.
- Hooks can reside in either local or server-side repositories, and they are only executed in response to actions in that repository.
- Hooks can be triggered pre or post commits, pre or post merges etc.
- To install a hook, simply go to the .git/hooks directory and remove the .sample extension from the desired file.
- The files in the .git/hooks directory represent most but not all of the available hooks.

CIS 191

Linux and Unix Skills

LECTURE 11

Lecture Plan

1. Compiling Files
2. Make
3. Package Management with APT

Compiling Files

Compilers

- A ***compiler*** is a program that translates code from one language, usually a high-level language, to another language, usually a lower level language, such as ***assembly code***, ***object code***, or ***machine code***.
- A compiler may translate code to an ***intermediate representation (IR)*** in between the high and low level languages; IRs are often referred to as ***bytecode***.
- IRs can be
  1. compiled to machine code that is run by the operating system, or
  2. executed directly by an interpreter often referred to as a ***process virtual machine***, or
  3. compiled to machine code during execution by an interpreter often referred to as a ***just-in-time*** or ***JIT*** compiler.
- Assembly, object code, and machine code are specific to a particular computer architecture and operating system, while intermediate representations are not.

GCC

- The ***GNU Compiler Collection*** (***GCC***) is a compiler system produced by the GNU Project supporting various programming languages.
- GCC supports compilation for C, C++, Objective-C, Objective-C++, Fortran, Ada, and Go.
- With no options, gcc compiles the specified files to an executable usually named a.out.
- **–S** : compile to assembly.
- **–c** : compile to object code.

Assembly Code

- ***Assembly code*** is any code written in a low-level programming language - an ***assembly language*** - in which there is a very strong correspondence between the instructions in the language and the architecture's machine code instructions, usually one assembly statement per machine instruction.
- Each assembly language is specific to a particular computer architecture, commonly called the ***instruction set architecture*** or ***ISA*** e.g. Intel and AMD processors typically use an x86 ISA while Fujitsu processors typically use a SPARC ISA.
- Assembly code is converted into object code by an ***assembler***.
- Examples of assembly languages include the GNU assembly (gas), netwide assembly (nasm), flat assembly (fasm) etc.

"Hello World" x64 Netwide Assembly Example

```
        global    _start

        section   .text
_start: mov       rax, 1                ; system call for write
        mov       rdi, 1                ; file handle 1 is stdout
        mov       rsi, message          ; address of string to output
        mov       rdx, 13               ; number of bytes
        syscall                         ; invoke operating system to do the write
        mov       rax, 60               ; system call for exit
        xor       rdi, rdi              ; exit code 0
        syscall                         ; invoke operating system to exit
        section   .data
message:  db        "Hello, World", 10    ; note the newline at the end
```

Object Code

- ***Object code*** is machine code that contains placeholders or offsets that are defined in other object and library files.
- Object code is itself not executable, but a ***linker*** combines several object and shared library files to generate an executable.
- An executable file may be ***dynamically linked***, meaning that the operating system links the shared libraries needed by an executable when it is executed, or ***statically linked***, meaning that the shared libraries needed by the executable are stored inside the executable file itself.
- The GNU linker ***ld*** combines a number of object and archive files, relocates their data and ties up symbol references e.g. the command

```
ld -o output /lib/crt0.o hello.o -lc
```

links the file /lib/crt0.o with the files hello.o and libc.a (which is found by searching the default library search directories), and outputs the linked file to the file output.

Linker Scripts

- The main purpose of a ***linker script*** is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file.
- Object files usually define the following sections:
  1. ***.text*** - the address of the code section,
  2. ***.data*** – initialized data i.e. variables that are not defined within a function or are defined in a function but are defined as static so they retain their address across subsequent calls.
  3. ***.bss*** - uninitialized data i.e. all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
  4. ***.heap*** – dynamically allocated data, usually using malloc, realloc, calloc, and free.
  5. ***.stack*** – the call stack.

Linker Script Example

- The following linker script sets the entry point of the program to the symbol _start, and sets the address of .text section to 0x10000, .data to 0x8000000, and .bss to the address immediately after the .data:

```
ENTRY(_start)

SECTIONS

{

  . = 0x10000;

  .text : { *(.text) }

  . = 0x8000000;

  .data : { *(.data) }

  .bss : { *(.bss) }

}
```

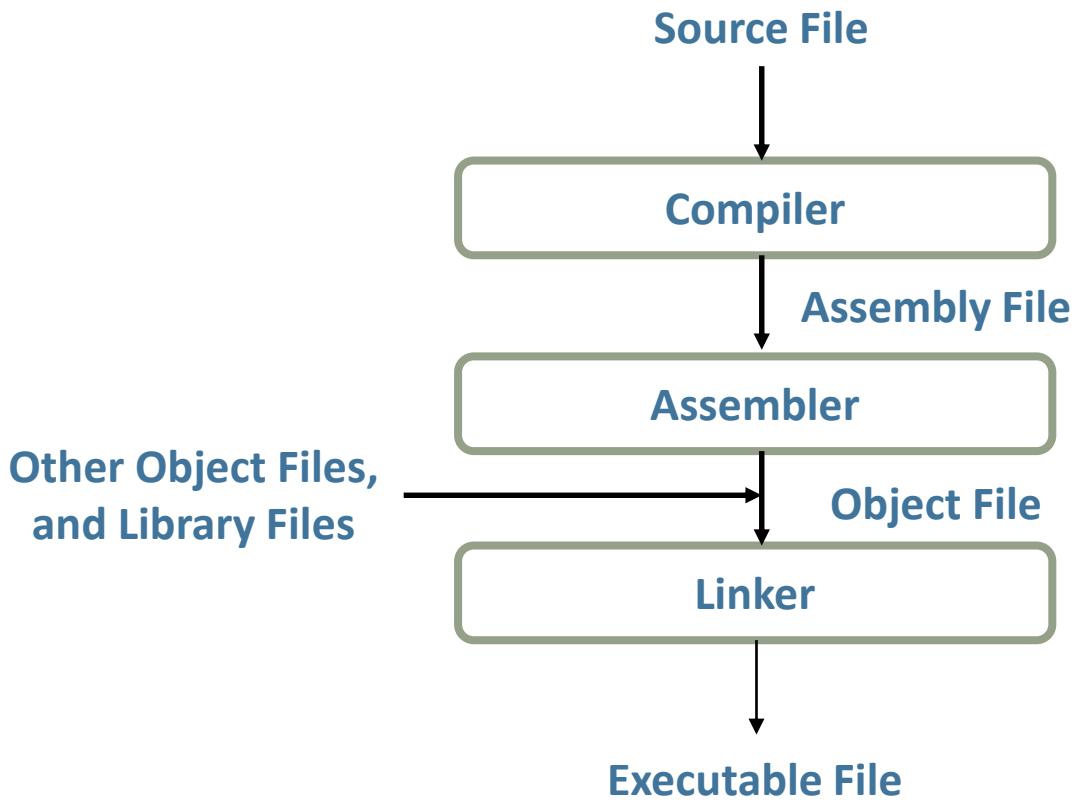
Machine Code

- ***Machine code*** is a strictly numerical language that can be executed directly by a computer's central processing unit e.g. in MIPS, the following instruction adds the contents in registers 1 and 2 and places the result in register 6:

```
[  op   | rs   |  rt | rd   |shamt| funct ]
    0      1      2      6      0      32      decimal
 000000 00001 00010 00110 00000 100000   binary
```

- Machine code typically occurs in a ***container format*** – e.g. ELF on Unix like systems, PE on Windows, Mach-O on macOS - that gives structure to the generated machine code, for example by dividing it into sections or specifying the architecture on which the executable is meant to be run on.
- Machine code is specific to the ISA of the architecture in which it is intended to be run.

Compilation Process



Make

Make

- ***Make*** is a tool that controls the generation of executables and other non-source files of a program from the program's source files.
- Make builds your program using a file called the ***makefile***, which lists each of the non-source files and how to compute it from other files.
- If you change a few source files and then run Make, it does not need to recompile all of your program, but instead updates only those non-source files that depend directly or indirectly on the source files that you changed.
- Make enables the end user to build and install your package without knowing the details of how that is done, because these details are recorded in the makefile that you supply.

Makefiles

- A simple makefile consists of ***rules*** with the following shape:

```
targets : prerequisites
        recipe
```

- A ***target*** is the name of a file that is generated by a program; usually executable or object files.
- A target can also be the name of an action to carry out, such as 'clean'.
- A ***prerequisite*** is a file that is used as input to create the target.
- A ***recipe*** is an action that make carries out.
- A recipe may have more than one command, either on the same line or each on its own line.
- **CAUTION!** You need to put a tab character at the beginning of every recipe line!

Makefile Example

```
mainfile : program.o file1.o file2.o
	gcc -o mainfile program.o file1.o file2.o
program.o : program.c
	gcc -c program.c
file1.o : file1.c file1.h
	gcc -c file1.c
file2.o : file2.c file2.h
	gcc -c file2.c
clean :
	rm mainfile program.o file1.o file2.o
```

Make

- When you execute the command make in the command-line, Make reads the makefile in the current directory, usually a file named makefile or Makefile, and makes the first target whose name does not start with a '.'; this is called the ***default goal***.
- You can change the default goal by setting the .DEFAULT_GOAL special variable e.g. to set the default goal to the target 'all' set

```
.DEFAULT_GOAL := all
```

- You can also specify which target Make should process by providing command line arguments e.g. the following command makes the target 'install':

```
make install
```

- When making a target, Make first processes the rules on which the target depends.
- Make uses the last-modification times of targets to decide which of the files need to be updated, and hence will not remake a target if it and its prerequisites exist and have not been updated.

Phony Targets

- A ***phony target*** is a target that is not the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request e.g. the phony target clean below removes all object files:

```
clean:
        rm *.o temp
```

- Since the clean has no prerequisites, it will not work properly if a file named clean is ever created in this directory as clean would always be considered up to date and its recipe would not be executed.
- To avoid this problem you can explicitly declare the target to be phony as follows:

```
.PHONY: clean
clean:
        rm *.o temp
```

- Once this is done, 'make clean' will run the recipe regardless of whether there is a file named clean.

Makefile Variables

- ***Makefile variables*** allow a text string to be defined once and substituted in multiple places e.g.

```
objects := file1.o file2.o

mainfile : $(objects)

	gcc -o mainfile $(objects)

...
```

- To substitute a variable's value, write a dollar sign followed by the name of the variable in parentheses or braces: either '$(foo)' or '${foo}' is a valid reference to the variable foo.
- Because of the use of the dollar sign '$' for variable substitution by Make, you must write '$$' to have the effect of a single dollar sign in a file name or recipe.

Pattern Rules and Automatic Variables

- A ***pattern rule*** contains the character '%' (exactly one of them) in the target; otherwise, it looks exactly like an ordinary rule.
- The target is a pattern for matching file names; the '%' matches any nonempty substring, while other characters match only themselves e.g. '%.c' as a pattern matches any file name that ends in '.c'. 's.%.c' as a pattern matches any file name that starts with 's.', ends in '.c' and is at least five characters long.
- ***Automatic variables*** are special variables that have values computed afresh for each rule that is executed, based on the target and prerequisites of the rule.
- $@ : The file name of the target of the rule.
- $< : The name of the first prerequisite.
- $? : The names of all the prerequisites that are newer than the target, with spaces between them.
- $^ : The names of all the prerequisites, with spaces between them.
- $* : The stem with which a pattern rule matches e.g. 'foo' in the target 'a.foo.b' with pattern 'a.%.b'

Pattern Rules and Automatic Variables Example

- The rule below makes any 'x'.o file from an 'x'.c, for any non-empty string 'x':

```
%.o : %.c
    gcc -c $< -o $@
```

- The recipe uses the automatic variables '$@' and '$<' respectively to substitute the names of the target file and prerequisite source file in each case where the rule applies.

Make String Processing Functions

- ***$(patsubst pattern,replacement,text)*** finds whitespace-separated words in *text* that match *pattern* and replaces them with *replacement* e.g. the following piece of code demonstrates how to use pattern substitution to define a list of object files from their corresponding source files:

```
CFILES := file1.c file2.c file3.c file4.c file5.c file6.c file7.c file8.c

OFILES := $(patsubst %.c,%.o,$(CFILES)).
```

- ***$(strip string)*** removes leading and trailing whitespace from *string* and replaces each internal sequence of one or more whitespace characters with a single space e.g. '$(strip a b c)' evaluates to 'a b c'.

- ***$(filter pattern…,text)*** returns all whitespace-separated words in *text* that do match any of the *pattern* words, removing any words that do not match e.g. '$(filter %.c %.s,foo.c bar.c baz.s ugh.h)' evaluates to 'foo.c bar.c baz.s'.

- ***$(filter-out pattern…,text)*** returns all whitespace-separated words in *text* that do not match any of the *pattern* words, removing the words that do match one or more.

- And others!

configure, make, make install

- Many free software packages can be built from source using the three commands:

```
./configure
```

```
make
```

```
make install
```

- ./configure runs the configure shell script that makes sure all of the dependencies for the rest of the build and install process are available, and finds out whatever it needs to know to use those dependencies, and then builds a new Makefile.
- Once configure has done its job, you can invoke make to build the software.
- Finally, the make install command will copy the built program, and its libraries and documentation, to the correct locations.

Package Management with APT

Package Managers

- A ***package manager*** or ***package-management system*** is a collection of software tools that automates the process of installing, upgrading, configuring, and removing computer programs for a computer's operating system in a consistent manner.
- A ***package*** file is a compressed collection of files that may consist of numerous programs and data files that support the programs, metadata about the package, pre- and post-installation scripts that perform configuration tasks before and after the package installation etc.
- Packages are made available to the users of a distro in central ***repositories*** that may contain many thousands of packages, each specially built and maintained for the distro.

Package Managers

- Package management can occur at the distro level e.g. dpkg for Ubuntu, homebrew for OS X, or at the application level e.g. OPAM for OCaml, pip for Python, Maven for Java etc.
- Some package managers have higher level ***frontend package managers*** that simplify the functionality of the base package manager for ease of use e.g. apt for dpkg, yum for RPM etc.
- Common package management tasks include finding a package in a repository, installing a package, removing a package, updating packages from a repository, upgrading a package from a package file, displaying information about an installed package etc.

APT

- ***Advanced Package Tool***, or ***APT***, is a set of tools for managing Debian packages.
- ***apt-get*** and ***apt-cache*** are a command-line based front ends developed by the APT project.
- ***apt*** is a second command-line based front end provided by APT that enables some options better suited for interactive usage by default compared to more specialized APT tools like apt-get and apt-cache.
- The APT project also spawned various other command-line and graphical front-ends, including synaptic, aptitude (which includes both a text mode interface and a graphical interface), wajig, etc.

APT Sources

- APT tools work with ***sources***, repositories that publish Debian packages.
- The file /etc/apt/sources.list lists the different official sources that publish Debian packages, and the directory /etc/apt/sources.list.d stores additional sources
- The ***add-apt-repository*** command adds a repository into the /etc/apt/sources.list or /etc/apt/sources.list.d, or removes an existing one e.g. the following command adds the MongoDB repository to the source.list file:

```
sudo add-apt-repository 'deb [arch=amd64]
https://repo.mongodb.org/apt/ubuntu bionic/mongodb-org/4.2 multiverse'
```

- Ubuntu provides a platform called Launchpad that enables software developers to create their own repositories called ***Personal Package Archives*** or ***PPAs***.
- You can add PPAs to your sources.list.d directory via the add-apt-repository command e.g. the following command adds the unstable Kubuntu Continuous Integration PPA to the sources.list.d directory:

```
sudo add-apt-repository ppa:kubuntu-ci/unstable
```

apt commands

- ***apt update*** and ***apt-get update*** download package information from all configured sources.
- ***apt upgrade*** and ***apt-get upgrade*** install available upgrades of all packages currently installed from the sources configured via sources.list.
- ***apt full-upgrade*** and ***apt-get dist-upgrade*** perform the function of upgrade but will remove currently installed packages if this is needed to upgrade the system as a whole.
- ***apt install*** package and ***apt-get install*** package install package.
- ***apt remove*** package and ***apt-get remove*** package remove package, but leaves its configuration files in the system in case the package was removed by mistake.
- ***apt purge*** package and ***apt-get purge*** package removes package as well as its configuration files.

apt commands

- ***apt reinstall*** package and ***apt-get reinstall*** package reinstall package.
- ***apt autoremove*** and ***apt-get autoremove*** remove packages that were automatically installed to satisfy dependencies for other packages and are now no longer needed.
- ***apt search*** regex and ***apt-cache search*** regex searches for the given regex term(s) in the list of available packages and displays matches.
- ***apt show*** package and ***apt-cache show*** package show information about the given package(s) including its dependencies, installation and download size, sources the package is available from, the description of the packages content and much more.
- ***apt list*** [OPTION] pattern displays a list of packages whose package names match the globbing pattern pattern.
  - The option to apt list can specify packages that have been installed (**--installed**), upgradeable packages (**--upgradeable**) or all available package version (--**all-versions**) versions.

CIS 191

Linux and Unix Skills

LECTURE 12

Lecture Plan

How Linux Works I

1. Hardware Overview
2. Process Management
3. The Virtual Filesystem

Kernel Roles

- The ***kernel*** is the core of the operating system, with complete control over everything in the system.
- The main roles of the kernel are:
  1. ***Process Management*** - the kernel enables multiple user applications to run at the same time.
  2. ***Memory Management*** - the kernel allocates memory to processes, reclaims memory from dead processes, and prevents processes from interfering with each others memory.
  3. ***Device Management*** - the kernel makes hardware devices available to processes, and prevents them from misusing the devices.
  4. ***Inter-process Communication*** – the kernel enables processes to communicate with each other in a safe way.

Hardware Overview

The CPU and Memory

- The instructions of a program (i.e. machine code) are executed by the ***central processing unit*** (CPU).
- The instructions of a program, as well the state needed to execute the machine code, are stored in ***memory***.
- The CPU also has a set of ***registers*** that maintain part of the state needed to execute machine code.
- To execute a program, the CPU fetches the instruction from memory, gathers data needed to actually execute the instruction such as the instruction type or data from the register file, executes the instruction, stores the result in memory or the register file, and then repeats this cycle.

The Register File

- Some of the registers in the register file have a special usage.
- The ***instruction pointer*** or ***program counter*** register stores the address of the next instruction to be executed.
- The ***stack pointer*** register stores the address of top of the stack.
- The ***stack base pointer*** register stores the address of the base of the stack.
- The ***link register*** stores the return address.
- ***Control registers*** change the behaviour of the CPU e.g. in x86 the CR3 control register stores the address of the page directory used when virtual addressing is enabled by setting the PG bit of the CR0 control register.

Interrupts

- An ***interrupt*** is an input signal to the processor indicating an event that needs immediate attention.
- The processor responds by suspending its current activities, saving its state, and executing a function called an ***interrupt handler*** or ***interrupt service routine*** (***ISR***) in response to the interrupt.
- A ***hardware interrupt*** or ***interrupt request*** (***IRQ***) is an interrupt signalled by a hardware device to communicate that it needs attention from the operating system e.g. pressing a keyboard key.
- A ***software interrupt*** is an interrupt signalled by software running on a CPU to indicate that it needs the kernel's attention e.g. an ***exception*** is a software interrupt.
- Hardware interrupts are ***asynchronous*** i.e. they can happen at any point in the execution of a program, while software interrupt and exceptions are ***synchronous*** i.e. they can only occur at certain points while the program is executing.

The Interrupt Vector Table

- Most architectures implement an ***interrupt vector table*** or ***IVT*** is a data structure that maps interrupt requests to interrupt handlers.
- Each entry of the interrupt vector table, called an ***interrupt vector***, is the address of an interrupt handler that will be automatically invoked by the CPU when the corresponding interrupt occurs.
- Some architectures (e.g. MIPS) do not have an interrupt vector table; instead, the processor stores the cause of the interrupt in a special register, after which an exception handler responds to the interrupt.
- Initializing the interrupt vector table is one of the first things that the kernel does upon booting.

Processor Modes

- ***Processor modes*** are operating modes for the central processing unit of some computer architectures that place restrictions on the type and scope of operations that can be performed by certain processes being run by the CPU.
- This design allows the operating system to run with more privileges than application software.
- Only highly trusted kernel code is allowed to execute in the unrestricted mode, often called ***kernel mode***.
- User code runs in a restricted mode, often called ***user mode***, and must use a ***system call*** to request the kernel perform on its behalf any operation that could damage or compromise the system.

Call Stacks

- ***Call stacks*** stores information about the active subroutines of a program.
- The primary purpose of a call stack is to store ***return addresses***.
- For example, if a routine f calls a routine g when executing, g must know where to return when its execution completes.
- To accomplish this, the address following the instruction that jumps to g, the return address, is pushed onto the call stack with each call.
- The call stack often also stores some more state, such as the local variables used in a routine, the parameters that were used when the routine was called, etc.
- Many ISAs provide special instructions for manipulating call stacks.

Stacks on Stacks on Stacks on Stacks

- In Linux, each process has a ***user stack*** that is used when the process is running in user mode, and a ***kernel stack*** that is used when the process executes a system call in kernel mode.
- The kernel stack is also used when the kernel is running an interrupt handler.
- The kernel stack has a fixed, relatively small size of 8KB.
- Linux may also be configured to use a smaller 4KB kernel stack, in which case the kernel uses two additional 4KB stacks: a ***hard IRQ stack*** for handling interrupts, and a ***soft IRQ stack*** for handling deferrable functions.
- If they exists, there is only one hard IRQ stack and one soft IRQ stack per CPU; these two stacks are not allocated for each process.

Process Management

Processes

- A ***process*** is an instance of a running program.
- Each process has a ***process descriptor*** that stores all of the information related to a single process e.g. the process's priority, whether it is running on a CPU or blocked on an event, what address space has been assigned to it, which files it is allowed to address, and so on.
- The kernel maintains a doubly linked list called the ***process list*** that links together all existing process descriptors.

Lightweight Processes

- ***Lightweight processes*** enable multithreaded programs.
- Two lightweight processes may share some resources, like the address space, the open files, and so on.
- Each lightweight process is a distinct scheduling entity, and Linux associates a different PID with each lightweight process in the system.
- Linux supports ***thread groups***, where a thread group is a set of lightweight processes that implement a multithreaded application and act as a whole with regards to some system calls such as getpid(), kill(), and _exit().

Kernel Threads

- Traditional Unix systems delegate some critical tasks such flushing disk caches, swapping out unused pages, servicing network connections, etc to intermittently running lightweight processes called ***kernel threads***.
- In Linux, kernel threads differ from regular processes in that kernel threads run only in Kernel Mode, while regular processes run alternatively in Kernel Mode and in User Mode.
- The ancestor of all processes, called process 0, the ***idle process***, or, for historical reasons, the ***swapper process***, is a kernel thread created from scratch during the initialization phase of Linux that becomes the root of all kernel threads.
- Process 0 then creates another kernel thread named process 1, more commonly referred to as the ***init process***, which completes the initialization of the kernel before becoming a regular user process that becomes the root of all user processes.

System Calls

- The ***system call*** is the fundamental interface between an application and the Linux kernel.
- When a process running in User Mode invokes a system call, the CPU switches to Kernel Mode and starts the execution of a kernel function.
- When a system call is invoked, the CPU jumps to a function called the ***system call handler***.
- A process running in User Mode passes the system call arguments, including the ***system call number*** used to identify the required system call, by writing them to CPU registers.
- The system call handler saves the contents of most registers in the Kernel Mode stack and handles the system call by invoking a corresponding C function called the ***system call service routine***, before exiting by loading the registers with the values saved in the Kernel Mode stack and switching the CPU back from Kernel Mode to User Mode.

Creating and Destroying Processes

- The ***clone()*** system call creates a new process by duplicating the calling process.
- The caller can control whether or not the parent and child process share the same virtual address space, the table of file descriptors, and the table of signal handlers.
- The ***fork()*** system call invokes clone() and specifies that the parent process and the child process use different virtual address spaces.
- fork() is normally followed by a call to one of the ***exec()*** system calls which replace the current program with a new program.
- The ***exit_group()*** system call terminates a full thread group and the ***_exit()*** system call, terminates a single process, regardless of any other process in the thread group of the victim.

Pre-emptive Multitasking

- Linux achieves the effect of an apparent simultaneous execution of multiple processes by switching from one process to another in a very short time frame; this is called ***pre-emptive multitasking***.
- Pre-emptive multitasking involves the use of an interrupt mechanism which suspends the currently executing process and invokes a scheduler to determine which process should execute next e.g. older Intel systems have the Programmable Interval Timer (PIT) chip that can be used to generate an interrupt periodically after a fixed time interval.

Process Scheduling

- Linux can be configured to run with various scheduling algorithms, the default of which is CFS, the "Completely Fair Scheduler".
- The data structure used for the scheduling algorithm is a red-black tree.
- When the scheduler is invoked to run a new process, the operation of the scheduler is as follows:
  1. The leftmost node of the scheduling tree is chosen (as it will have the lowest spent execution time), and sent for execution.
  2. If the process simply completes execution, it is removed from the system and scheduling tree.
  3. If the process reaches its maximum execution time or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its new spent execution time.
  4. The new leftmost node will then be selected from the tree, repeating the iteration.

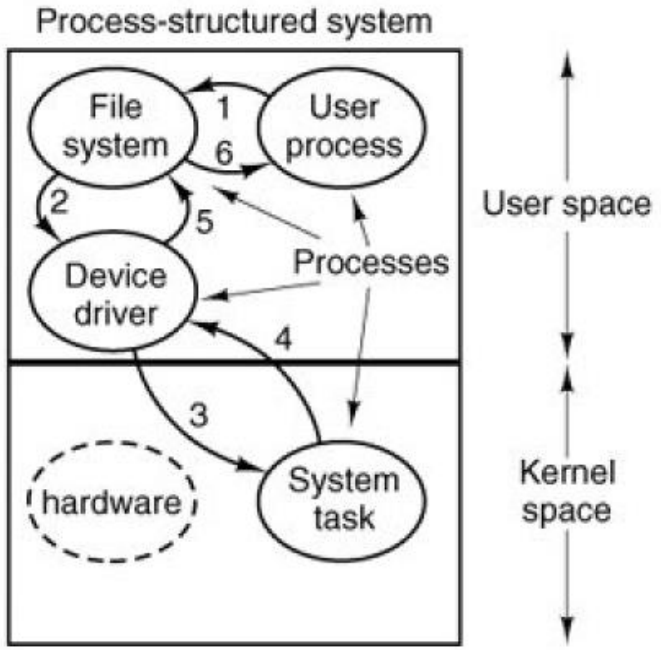
The Context Switch

- Every running process runs maintains some state in which it needs to run called its ***execution context***.
- The set of data that must be loaded into the registers before a process runs on the CPU is called its ***hardware context***.
- In Linux, a part of the hardware context of a process is stored in the process descriptor, while the remaining part is saved in the Kernel Mode stack.
- A ***context switch*** (or ***task switch***/***process switch***) occurs when the kernel suspends the execution of the process *P* running on the CPU and resumes the execution of some other previously suspended process *P′* i.e. a context switch is the activity consisting of saving the hardware context of *P* and replacing it with the hardware context of *P′.*
- In Linux, a context switch essentially consists of two steps: switching the page table to install a new address space and switching the Kernel Mode stack and the hardware context.

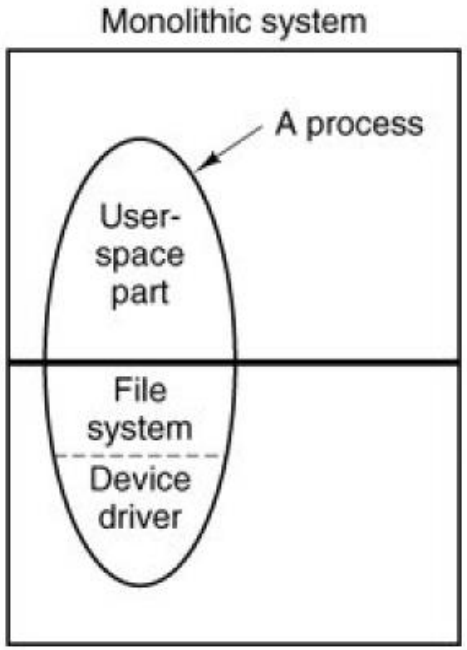
Microkernels vs Monolithic Kernels

- Context switches are usually computationally intensive, and much of the design of operating systems is to optimize the use of context switches.
- Linux is a ***monolithic kernel***, meaning that most of the kernel services are performed when a process is running in Kernel Mode, which does not require a context switch, and hence boosts the performance of the system as a whole.
- In contrast, a ***microkernel*** only provides basic services like memory allocation, scheduling and inter process communication, and runs other services like the filesystem and device drivers as user processes called ***servers***; user space programs receive services from servers through inter-process communication.
- Monolithic kernels tend to be faster than microkernels, largely due to fewer context switches, while microkernels are smaller, simpler, more modular, and tend to be more secure than monolithic kernels.

Microkernels vs Monolithic Kernels



1–6 are request and reply messages between four independent processes.



The user-space part calls the kernel-space part by trapping. The file system calls the device driver as a procedure. The entire operating system is part of each process

Inter-process Communication

- Linux provide many methods for processes to communicate with each other:
  - ***Signals*** which are used to notify processes of change in states or events that occur within the kernel or other processes.
  - ***Anonymous pipes*** and ***named pipes*** which provide a mechanism for one process to stream data to another.
  - ***Message queues*** which enable a process writes a message packet on the message queue to another process before exiting.
  - ***Sockets*** which provide a mechanism for processes to communicate using the Client-Server architecture.
  - ***Shared memory*** which is memory that may be simultaneously accessed by multiple processes.
  - ***Semaphores***, ***mutexes***, ***futexes***, and ***file locks*** which provide a locking and synchronization mechanism for processes sharing resources
  - etc!

The Virtual Filesystem

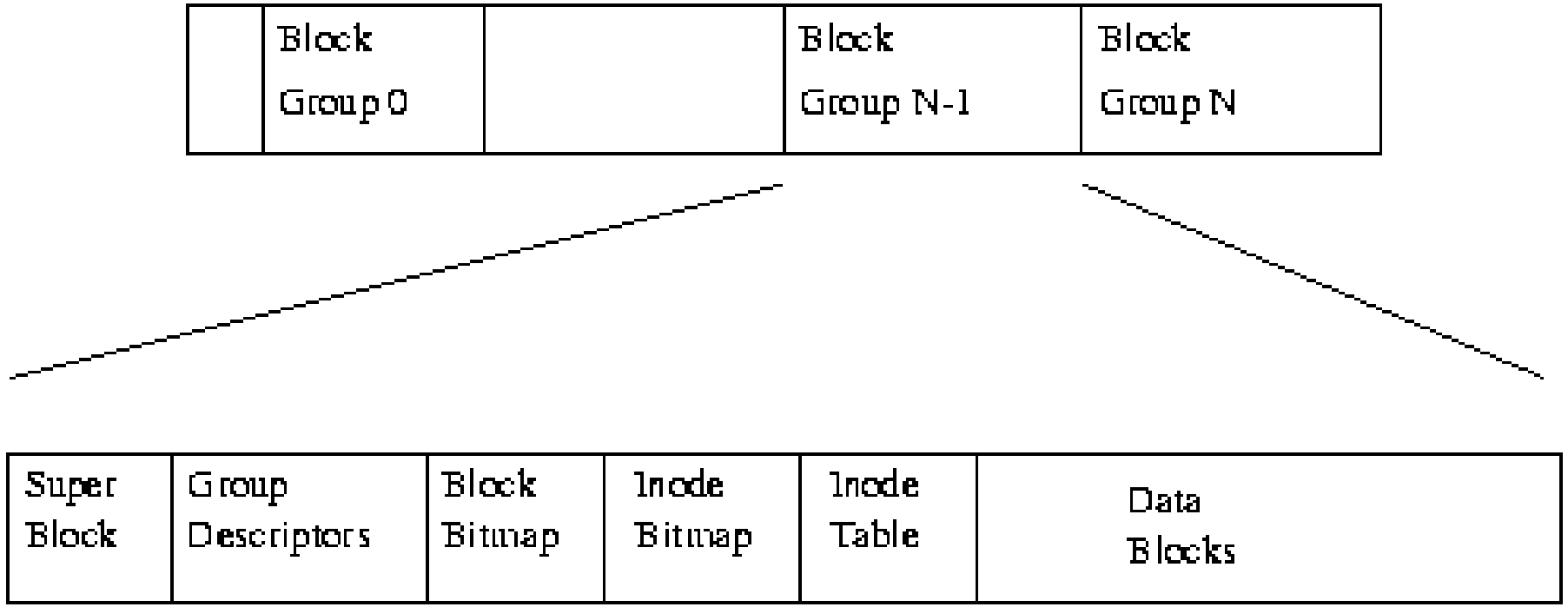
Filesystems

- A ***filesystem*** organizes data in a storage device in a way that makes it easy to store and retrieve data from the device.
- Examples of filesystems are the Ext family used in Linux, the FAT family used in DOS and Windows, NTFS used by newer versions of Windows etc.
- Some file systems are used on local data storage devices; others provide file access via a network protocol.
- Data storage devices often have ***partitions***, and the separate partitions may have different filesystems on them; when dual-booting from the a hard disk, one operating system uses one partition while the other uses another.
- The disk stores the information about the partitions' locations and sizes in an area known as the ***partition table***, which is often in the first sector of the disk.

Filesystem Example – Ext2

- The Second Extended Filesystem (Ext2) uses
  1. ***blocks*** as the basic unit of storage,
  2. ***inodes*** as the mean of keeping track of files and system objects
  3. ***block groups*** to logically split the disk into more manageable sections,
  4. ***directories*** to provide a hierarchical organization of files,
  5. ***block*** and ***inode bitmaps*** to keep track of allocated blocks and inodes, and
  6. ***superblocks*** to define the parameters of the file system and its overall state.

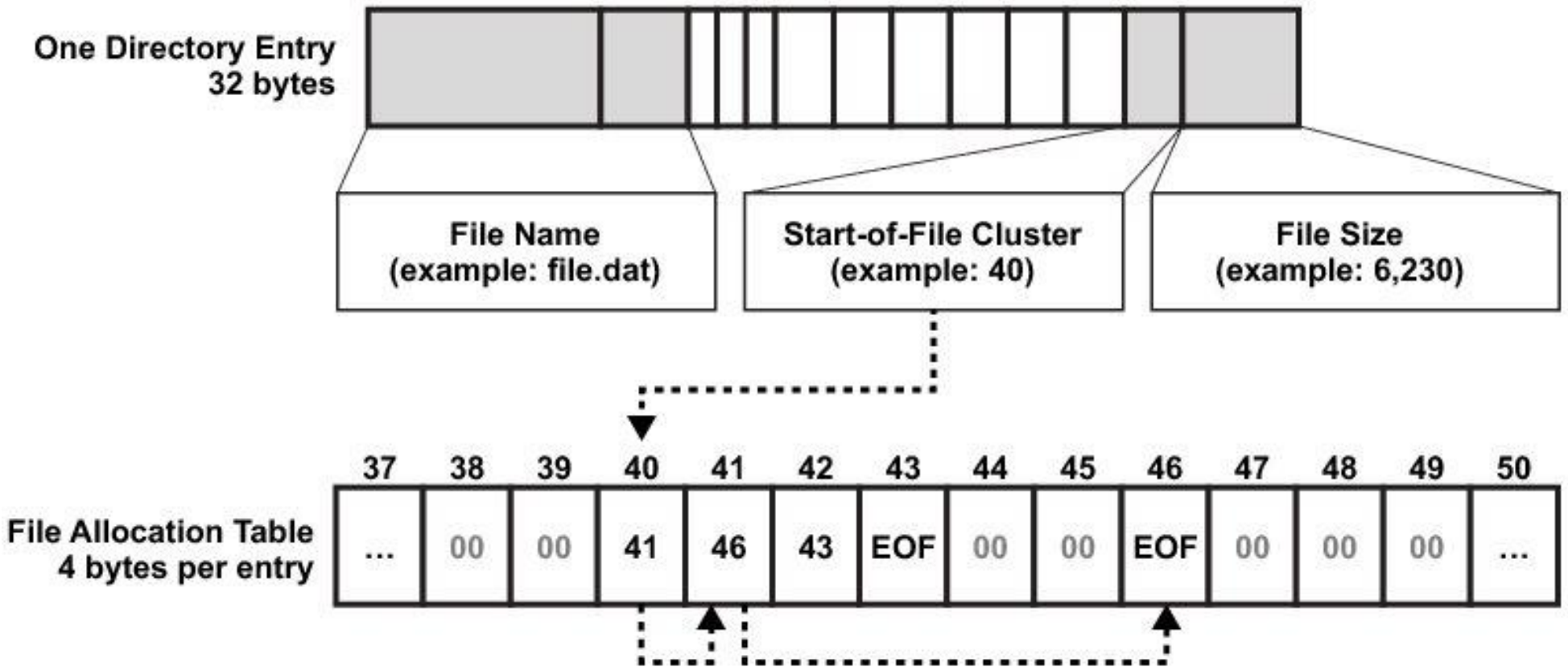
Ext2 File-system



Filesystem Example – FAT12

- A FAT12 filesystem consists of four major sections: the ***boot sector***, ***FAT tables***, ***root directory***, and ***data area***.
- The ***boot sector*** consists of the first sector on the disk and contains specific information about the rest of organization of the file system.
- ***FAT tables*** contain pointers to every cluster on the disk, and indicate the number of the next cluster in the current cluster chain, the end of the cluster chain, whether a cluster is empty, or has errors.
- The ***root directory*** is the primary directory of the disk; each sector of a directory contains 16 directory entries, and each directory entry describes and points to some file or subdirectory on the disk.
- The ***data area*** contains file and directory data and spans the remaining sectors on the disk.

FAT File-System



The Virtual Filesystem

- The Linux kernel implements a ***virtual filesystem*** or ***VFS*** that provides a standard Unix file system call interface to several kinds of filesystems.
- For example, the VFS enables a user to open a file that is stored in a USB disk formatted by the NTFS filesystem, and copy its contents into a file stored on a hard disk formatted by the Ext3 filesystem.
- The key idea behind the VFS consists of introducing a ***common file model*** capable of representing all supported filesystems.
- This model strictly mirrors the file model provided by the traditional Unix filesystem.

VFS System Call Examples

- ***chdir( )***, ***fchdir( )***, ***getcwd( )*** - Manipulate current directory
- ***open( )***, ***close( )***, ***creat( )***, ***umask( )*** – Open, close, and create files
- ***read( )***, ***write( )*** - Carry out file I/O operations
- ***mount( )***, ***umount( )*** - Mount/unmount filesystems
- ***link( )***, ***unlink( )***, ***rename( )*** - Manipulate directory entries
- ***readlink( )***, ***symlink( )*** - Manipulate soft links
- ***mkdir( )***, ***rmdir( )*** - Create and destroy directories
- ***chown( )*** - Modify file owner
- ***chmod( )*** - Modify file attributes
- ***truncate( )***, ***ftruncate( )***, ***truncate64( )***, ***ftruncate64()*** - Change file size

The Virtual Filesystem

- The common file model consists of four object types: ***superblock***, ***inode***, ***dentry*** (directory entry), and ***file***.
- A ***superblock*** object stores information about the whole filesystem, such as the type of the filesystem, a list of all inodes, the name of the device containing the superblock etc.
- An ***inode*** object stores information about a specific file e.g. the length of the file, the time of last access, the owning user and group of the file etc.
- A ***file*** object stores information about the interaction between an open file and a process e.g. the dentry object associated with the file, the user's UID and group ID etc
- A ***dentry*** object stores information about the linking of a directory entry (that is, a particular name of a file) with the corresponding file e.g. the filename, the dentry object of its parent directory, the list of subdirectory dentries if the dentry is a directory etc.

The Virtual Filesystem