UMass Lowell
Computer Science Department
Introduction to Linux
Linux is an operating system

- An operating system (OS) is the software that makes the computer hardware operate.
- The OS talks to the computer hardware (memory, disk, cpu, video, sound, network) in order to allow other programs (applications) be used by a user.

Examples of operating systems:
  - Microsoft Windows
  - Apple MAC OS X
  - Apple IOS
  - Android OS
  - Linux
Single User vs Multi User

Operating systems are either:

- Single User – only one person can use it at a time
  - Windows 7
  - Apple IOS

- Multi User – more than one user can use it
  - Linux
  - MAC OS X
CLOSED VS OPEN SOURCE

• There are 2 types of operating systems:

  – CLOSED SOURCE (PROPRIETARY)

  – OPEN SOURCE (SOMETIMES ‘FREE’)
CLOSED SOURCE

• **CLOSED** OS’s are ones where either they run only on specific hardware AND/OR the programs that make up the Operating System CANNOT be modified by anybody except for the manufacturer. 2 examples:
  – Microsoft Windows
  – Apple OS X

• Although anybody can write programs to run on these systems, only the manufacturer (Microsoft / Apple) can change the way the internal operating system works.
• If a bug or security issue exists within the operating system, the user must wait for the vendor to produce the patch.
• Only the manufacturer is allowed to see or view the source code to the operating system.
OPEN SOURCE

• **OPEN** OS’s are ones in which anybody who wishes to can change the way the operating system runs. Because of this they can be made to run on almost any type of hardware 2 examples:
  – Ubuntu Linux
  – FreeBSD
• Anybody can gain access to the source code that makes up an ‘open’ OS.
• Since the source can be accessed, anybody can learn how the system works and can modify the way the system works:
  – For example, if a user is not happy with the way a printer works, they could modify the driver.
• Since everybody can see the source code, it becomes easier for security holes to be **discovered** and to be **fixed**. If a security hole exists, anybody can write the code to fix it. This allows security patches to be release much faster than closed source.
• Open Source is sometimes referred to as ‘FREE’
FREE SOFTWARE

• Linux is often referred to as ‘free’. In this case, free does not necessarily mean that you pay $0.00 for it.

• Free software is that in which the end user is ‘free’ to modify it in any way possible, as long as those modifications do not become ‘closed source’ (not available to others).

• The FREE SOFTWARE FOUNDATION is the driving force behind this type of software/operating system.
Software Freedom

• As outlined by the ‘Free Software Foundation’, you have:
  – The freedom to run the program, for any purpose (freedom 0).
  – The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
  – The freedom to redistribute copies so you can help your neighbor (freedom 2).
  – The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.
Software Freedom

• You can SELL free software that you have modified as long as you allow others to modify it and treat it as ‘free’. It is not wrong to charge for (or buy) free software.

• It is copyrighted with ‘copyleft’ – meaning you can use it, change it, redistribute it, sell it as long as it maintains that copyright of freedom (GPL). It’s not public domain software – public domain means you can take it and modify it so nobody else can.

• Many companies are making profits off of ‘free’ software where the software is given away but they provide support and bug fixes for a fee.
Open Source / Free Programs

• Software that runs on an operating system (like MS Office, Firefox, Chrome) can also be closed vs open source. The same rules apply for software as they do for an OS.

• There are hundreds of examples of open source software running on closed/proprietary operating systems. There are also CLOSED source software packages that run on proprietary and NON proprietary operating systems.
What is Linux?

LINUX is FREE software made up of 2 things:

• **Kernel (Linux)**  
  Developed by Linus Torvalds. This is the software that talks directly to the hardware, for accessing memory, hard disk and other hardware devices. It is responsible for running the programs that the end user asks to have run.

• **Software (compiler, commands) (GNU)**  
  Core utilities for compiling, file manipulation, editing developed by the Free Software Foundation, headed by Richard Stallman.
Linux History

• Before Linux, there was UNIX.
• UNIX still exists, but Linux is more popular and more widely used.
• It’s important to remember that LINUX is not UNIX and vice versa.
• Linux is considered ‘UNIX-Like’ (it looks like UNIX).
UNIX History

• The UNIX operating system was developed by Bell Labs (was part of AT&T) in 1969 by Ken Thompson and Dennis Ritchie in order to allow them to play a game called space travel on a specific computer.

• Initially it was written in assembly language but was later re-written in the C language to allow it to be ported to other types of computers. At that time it was used at some universities and businesses.

• AT&T’s licensing made it so that it (the source code) had to be given to anybody who asked for it.
BSD UNIX History

• A group known as CSRG at University of California Berkeley used ATT UNIX together with software and utilities they created to make an OS called the Berkeley Software Distribution (BSD).

• The CSRG group also created TCP/IP as part of a project in which the government needed to upgrade its defense network known as ARPANET. TCP/IP is what the internet operates with.
UNIX History

• When AT&T and Bell Labs split up, Bell Labs made UNIX a commercial product and ‘closed the source code’.

• Many companies licensed the UNIX product to run on their own workstations and hardware. SUN (now ORACLE) was an example of one of these companies, where they included ATT UNIX and BSD software together to form SUNOS.

• AT&T commercialized their version of UNIX called System III (later System V) and included BSD tools with it, such as VI, CSH and TCP/IP.
MINIX / FREEBSD History

• A UNIX-like version called MINIX was developed and was freely available (including the source code) to Universities, but not for commercial use.

• When the CSRG (BSD) group was losing their funding to continue the development of BSD, they decided to release their source code to the public. The AT&T UNIX license was taken out of it due to licensing issues. This forced a group to add in that important missing piece, and this is where the current FreeBSD and NetBSD UNIX-like software started from.
Linux History

• During this time, 2 separate projects together combined to create what we now know as LINUX:
  – Linus Torvalds KERNEL
  – GNU Software

• Computer enthusiasts who wanted more control over how their computers worked create both of these pieces.
Linus Torvalds Kernel

1. The ‘Linus Torvalds’ Kernel

   – **Linus Torvalds** (a Computer Science student in Finland) created the first version of what he called ‘Linux’. It consisted mainly of the kernel, with a few other programs (like the shell interpreter) that allowed people to ‘use’ the computer the kernel was running on.

   • The kernel in Linux is software that controls the hardware. It talks to the different components of a computer, such as:
     – Memory
     – Disk storage
     – Video
     – Sound
     – Network

   – Users do not directly interact with the kernel, they use programs which the kernel runs.
GNU Software

2. GNU Software:

• A college student named Richard Stallman (MIT) decided to create his own version of ‘UNIX’ like software (and kernel) after being continuously frustrated with the inability to modify the way commercial programs ran.
• He called his project GNU (GNU’s Not Unix).
• This project consisted of a kernel called GNU Hurd and more importantly applications such as compilers, text editors (including emacs), file utilities, mail readers, games.
• Many of these applications already existed on other versions of UNIX but Stallman re-wrote them (and the GNU Hurd kernel) under his new ‘Free Software Foundation’ license/copyright.
Early on, Stallman decided to use the kernel from Torvalds instead of his HURD kernel to run his GNU Software suite. Thus became LINUX as we know it now.

The Linux Kernel was brought into the GNU Public License, along with the GNU software – making it ‘free software’.

Primarily all distributions of linux (Red Hat, CentOS, Ubuntu, Debian, Fedora) are really a combination of the LINUX Kernel and GNU Software.

When referring to LINUX, it should rightfully be called ‘GNU/Linux’.

GNU Software
Happy 20\textsuperscript{th} Birthday Linux!

• 20 Years of Linux!
ACCESSING LINUX AT UML CS

- SERVERNAME is cs.uml.edu

  - PUTTY FROM WINDOWS

  - SSH FROM LINUX / MAC OS X

  - LINUX WORKSTATION IN OLSSEN 308
The Command Line (SHELL)

The Command Line (shell)

• After logging into your account, opening up a terminal window on a linux system, or running the Terminal application in OS X you will be placed into a **command line session, known as the ‘shell’**. There are several versions of the shell:
  – Bourne Shell (sh)
  – C Shell (csh)
  – Korn Shell (ksh)
  – Bourne-Again Shell (bash)

  By default, your CS account will use the ‘bash’ shell.

• **The shell is where you type in commands to run programs.**
• In fact, the shell IS a program itself.
The Command Line (SHELL)

• The backspace/delete key can be used to erase an entry at the command line.
• The commands and file/directory names in Linux are CASE SENSITIVE, meaning that upper and lower case characters are treated independently of another. **Most commands are in lower case.**
• After typing in a command, press the return/enter key to submit the command to the shell for execution.
• The output of the command will appear on your screen, along with any errors that the command produces.
• The shell is a **command interpreter**. Its job is to look at what you typed in and execute it. It **interprets** the command, options, arguments and any special characters used.
• Typing in an invalid command will yield **command not found**.

**EXAMPLES OF COMMANDS**

```
ls -l
who
go
ps -ef
kill
```
Command Line Syntax

Command Syntax

• A shell command line consists of several items.

GENERAL SYNTAX OF A COMMAND:

\[ \text{command} \ [\ -\text{option} \ldots] \ [\ \text{argument} \ldots] \]

A single space is used between commands, options and arguments.

• COMMAND
  – This is the name of the program. In linux and unix, this is normally in lowercase and is usually cryptic. In some cases, just typing in a command and pressing ENTER will run the command, without any options or arguments (known as running in default mode).
Command Line Syntax

• **OPTIONS**
  – These are also known as switches. These modify the default operation of the command. Options are ‘optional’!
  – Options are preceded with a – character followed by a single character.
  – If giving multiple options to one command, they can be listed after each other, with only one – character needed.
  – Conversely, multiple **–option** strings can be specified with a space in between.
    • `ls -a`
    • `ls -a -R`
    • `ls -aR`
  – Options are case sensitive – many commands have options where the same character means something different based on the case.
Command Line Syntax

- ARGUMENTS
  - An argument is the extra information given to the command and is specific to the command.
  - Some options REQUIRE that an argument be given after the option and in many cases an argument can be given without an option.
  - Arguments are also case sensitive.
  - Arguments require spaces between them and the command / option.

  `ls -R directory1`
  `ls directory1 directory2 file1`
Command Recall

When using the BASH shell, you can recall the previous commands you have typed by using the UP ARROW. Press enter to execute the command when it appears on the command line.

When scrolling through the previous commands with the UP ARROW, you can use the DOWN ARROW to see more recent commands.

The shell uses a file called .bash_history in your home directory to store this history.
Command Execution/Processes

• When a user presses enter to execute a command, the shell starts the command as a new process, running it as a 'child' of itself.

• The parent 'shell' process waits until the child program is finished and then produces the shell prompt again.

• Each process on a Linux system has a parent, including your login shell. A 'master' process called 'init' is the first process to start when the system boots.

• Each process is assigned a unique process id (PID) and is 'owned' by the user who started it.
The `ps` command can be used to look at the processes running on the system you are logged onto.

**SYNTAX: ps [ -e –f -u ... ]**

Typing in just `ps` will show you just the processes that you are running.

Using `ps -f` will show you your processes but also more information such as the parent process id (PPID).

Using `ps -ef` will show you ALL of the processes running on the system.

Using `ps -f -u username` will show you all of the processes running by a specific user.
Command Execution/Processes

• EXAMPLE OF ps –f WHILE LOGGED INTO THE CS.UML.EDU LINUX CLUSTER
FOREGROUND VS BACKGROUND

• When a program is running and the shell prompt is not available, it is running in the **foreground**.

• If you would like to run multiple programs at the same time, consider running them in **background mode**. The login shell will be the parent of each program, but each one runs independently and at the same time.

• Append your command with the **&** character if you would like to run the program in the **background**. After you press enter, your program will start and you will see the shell prompt again.

```
$ program1 &
[1] 30475
$ ps
PID TTY TIME CMD
30370 pts/2 00:00:00 bash
30475 pts/2 00:00:00 program1
30477 pts/2 00:00:00 ps
```
The `jobs` command can be used to find out which processes are running in the background.

```
$ jobs
[1] Running  program1 &
[2] Running  program2 &
```

Use the `fg #` command to place that program back into the 'foreground' (i.e. `fg 2` to place program2 into the foreground).

When running processes in the background, their output will still be written to the screen. To control this, use `output redirection`.
FOREGROUND vs BACKGROUND

Pause a program

While a program in running in the foreground, you can PAUSE/ SUSPEND it by pressing the CTRL-z key sequence.

The command shell will appear after doing this. To allow the command to run while you run another program, type in the command `bg` (this stands for background).

To bring the program back to the foreground, type the command `fg`.

While the program is running, the output will continue to appear on the screen, unless it is redirected to a file when the program started.
Stopping a process

Processes can be stopped if the user owns the process running. The root user can stop any running process.

When a process is stopped, all of its children are also terminated. If root terminates all of the 'bash' processes, all the users would be logged out! When you log out, any processes you are running are normally terminated.

The kill command can be used to terminate processes:

**SYNTAX:** `kill [ -signal ] PID`

The signal is a number which tells the program how to terminate itself.

If no signal is used, this is a nice kill and will allow the program to close any open files and delete and temporary files it may have opened.

If the -9 signal is used, the program immediately stops without closing files, deleting temporary files, etc. This should only be used if 'kill' doesn't stop it.
Stopping a process running the foreground

Stop a program (nicely)
While a program is running, use the CTRL-c key sequence to terminate it.

Stop a program (harshly)
If a program will not terminate using CTRL-c, use the CTRL-\ sequence. This manner may not properly save any output the program was creating or close files that were open.

Pause the program output
To stop the output from scrolling across your screen, use the CTRL-s key sequence. To restart the output, press the CTRL-q key.
Realize that while the output is stopped, the program is continuing to run.
By default, the output of a program is sent to the screen that your shell is running in. This is known as standard output.

To send the output of a program to a file instead of the screen, use standard output redirection by using the > character.

BE CAREFUL when specifying the file to send the data to. If the file already exists, IT WILL BE OVERWRITTEN.

TO APPEND to an existing file, use the >> key sequence instead.

```bash
ps -ef > file1
ps -ef >> file1
```
STANDARD ERROR REDIRECTION

When a program produces errors, they get written to the ‘standard error’ channel. By default, standard error is the terminal screen.

To specify that ERRORS get sent to a file use the `2>` syntax:

```bash
ls missingfile 2> file1
```

To send **standard output and standard error to the same file**, use this:

```bash
ls goodfile missingfile > file1 2>&1
```
STANDARD INPUT / REDIRECTION

When a command runs that expects input, it uses the **STANDARD INPUT** shell subsystem. By default, standard input is via the keyboard – meaning that the program will wait until you type in some data and then use the EOF marker to indicate you are done.

You can instead **REDIRECT STANDARD INPUT** from a file instead of the keyboard by using the < symbol.

**SYNTAX:** command < filename

**EXAMPLE:**

$ tr a b < textfile

all a's changed to b's in the textfile
(tr command changes any occurrence of letter1 to letter2)

The CTRL-D keyboard sequence will indicate EOF (end of file) to a program reading standard input from the keyboard.
The shell PIPE subsystem can be used to send the output of one program as input into another program. You can string together multiple PIPES on one command line.

**SYNTAX:** command1  |  command2

**EXAMPLE:**

```
$ ps -f
UID   PID  PPID  C   STIME   TTY   TIME CMD
user  30370 30369  0      14:13  pts/2   00:00:00  -bash
user  30620 30370  0      15:09  pts/2   00:00:00  ps  -f
$ ps -ef  |  more
```

*This sends the output of the `ps -f` command to the `more` command (to see the output one page at a time)*

Pipes are useful with the **more and lpr commands** when running a program that produces output that you want to either print or view one page at a time on the screen:

```
$ program1  |  more
$ program1  |  lpr
$ ps -ef  |  grep user1  |  more
```

Files and Directories

Almost all of the commands in linux (and all unix-type systems) interact with either **files** or **directories**. In fact, most commands themselves are files.

Files in linux are organized into directories, where directories contain files and other directories.

The linux directory structure is shaped like an inverted tree, where the ROOT of the tree is at the top, and all branches represent directories.
Sample Linux Filesystem Layout
File and Directory names

- File and directory names are case-sensitive when referred to by the shell.
- Can have 2 files/directories with the same name in the same directory if at least one of the letter cases are different.
- Stay away from using special characters like these in file/directory names:
  \ / < > , * [ ] |
PATHNAMES

A directory and/or file is referenced by it’s path, which is a map to its location, either from the ‘current’ directory or from the root of the linux file system.

EXAMPLES:

/usr/local/bin  (directory path)
/usr/local/bin/ spacewar  (path to a file inside a directory)
directory1  (directory path)
directory1/myfile  (path to a file inside a directory)

Only certain parts of the linux system are ‘writable’ by users, where the others are writable by only the system account (known as the ‘root’ account), and special users who are part of the ‘root’ group.

PATHNAMES are case sensitive. (i.e. directory1 is different than Directory1)
Home Directory

Each user account is assigned a **home directory** which by default they have complete read and write access to.

**EXAMPLE:** `/usr/cs/undergrad/2015/joeuser`  
The ‘home directory’ for a user named joeuser
HOME DIRECTORY

When your home directory is created on the CS server, you have files already there (ls –a):

.bash_logout - contains commands that run when you logout
.bashrc - contains code to customize your shell session
.forward - contains the email address to send your CS email to
.kde - configuration directory used by linux workstations
.screenrc – configuration file for the screen program
.bash_profile - contains commands that run when you login
.emacs - configuration file for your emacs sessions
.gtkrc - configuration file used by linux workstations
.public_html - your personal CS website space (this is a directory)
THE CURRENT DIRECTORY

When logged into your Linux account, your shell is always ‘in a directory’, known as the ‘current directory’. By default, logging in places your shell session into your home directory as your current directory.

Your current directory will not change unless a command changes it.

The `pwd` command will show what your current directory is.
Changing your current directory

To access the files in a directory using the relative pathname, set your current directory (cwd) to the directory the files are in, using the cd command.
The *cd* command

cd command:

This command stands for ‘change directory’ and is used to change your current directory to the pathname listed after it. In some cases, you may not be able to ‘cd’ into a directory, if permissions don’t allow.
The *cd* command

**SYNTAX:** cd pathname

- cd directory1  
  relative path, assumes directory1 is under cwd
- cd ~/directory1  
  directory1 under your home directory
- cd ~user1/directory2  
  directory2 under user1’s home directory
- cd /directory1  
  Refers to directory1 under the SYSTEM root!
- cd  
  Sets the current directory to the user’s home directory

*USING THE cd COMMAND WITHOUT ANY ARGUMENTS AFTER IT WILL CHANGE YOUR CURRENT DIRECTORY TO YOUR HOME DIRECTORY*
The *pwd* command

**pwd command:**  
This command will display your current directory

**SYNTAX: pwd**

```bash
$ pwd  
/usr/cs/undergrad/2015/joeuser
```

```bash
$ cd directory1
$ pwd
/usr/cs/undergrad/2015/joeuser/directory1
```
Absolute vs Relative Paths

When a command refers to a file or directory, the pathname is either an absolute or relative path.

The user can decide which path to use.

The decision on which to use is normally based on where the user’s session is in the directory tree.
ABSOLUTE PATH

ABSOLUTE (full path name)

This is the path to the object starting at ‘root’ (/)

/usr/cs/undergrad/2015/joeuser/directory1/filea
/usr/local/bin
A relative pathname is indicated based on where your current directory is.

`directory1/filea`

Is a reference to a file ‘filea’ under the directory ‘directory1’. This would wonly be valid if there is a directory called ‘directory1’ under your current directory.
If a pathname is prefixed with the `~/` characters, this is referring to a path starting at the home directory of the user logged into the shell. This will work regardless of what your current directory is.

`~/directory1/filea`
If a pathname is prefixed with `~username`, this is referring to a path starting at the **home directory** of the **user** specified. This will work regardless of what your **current directory** is.

`~professor/directory1/filea`
SEARCH PATH

When a command is typed into the shell prompt, the shell uses the SEARCH PATH to locate the command typed in.

The search path is a list of directories for the system to search in to find the command requested.
SEARCH PATH

Example of a search path:
/usr/local/bin:/usr/bin:/bin:/usr/games

View the search path with `echo $PATH`

If the command is not in the search path, the `command not found` message will appear.
For security purposes, the current directory (.) is not in the search path.

To run a command (such as your compiled C code, a.out) that is in your current directory, type:

```
$ ./command
(i.e. ./a.out)
```
MODIFYING THE SEARCH PATH

The search path can be modified with this command:

\$ export PATH=$PATH:pathname-to-add

Example:

\$ export PATH=$PATH:~/bin

(this adds the bin directory under your home directory to the search PATH)

Using a text editor, place this command in your .bash_profile file to ensure it is saved for your next login.
Special Directories

Each directory contains 2 special directories:

- .  Refers to the current directory
- .. Refers to the directory ABOVE the current directory

These can be used in pathnames, as relative paths.

```
$ pwd
/usr/cs/undergrad/2015/joeuser
$ cd directory1
$ pwd
/usr/cs/undergrad/2015/joeuser/directory1
$ cd ..
$ pwd
/usr/cs/undergrad/2015/joeuser
$ cd directory1
$ pwd
/usr/cs/undergrad/2015/joeuser/directory1
$ cd .
$ pwd
/usr/cs/undergrad/2015/joeuser/directory1
$ cd ~janeuser
$ pwd
/usr/cs/undergrad/2015/janeuser
```
Manipulating Directories

Directories can be created, deleted, moved, copied, renamed and listed.

You can do these operations on DIRECTORIES if you ‘own’ them OR you have write permission for them (the default if you created them).
Creating a directory

**mkdir command:** This command creates a directory with the pathname following it. This command will fail if a directory with the same name already exists. You CAN have multiple directories with the same name if they are on a different path.

**SYNTAX:** `mkdir < -p > directory-pathname`
- The directory-pathname cannot already exist.
- **REMEMBER:** pathnames are CASE SENSITIVE
- **The option** `-p` can be used to create a directory in a path if the directory doesn’t already exist.
Creating a directory examples

```
$ cd
$ mkdir directory1
$ mkdir Directory1
$ mkdir directory1/directory2
$ mkdir directory1
mkdir: cannot create directory `directory1': File exists
$ mkdir directory1/directory2/directory3/directory4
mkdir: cannot create directory `directory1/directory2/directory3/directory4': No such file or directory
$ mkdir –p directory1/directory2/directory3/directory4
```
Listing a directory’s contents

**ls command:** This command will display the contents of a directory

**SYNTAX:** `ls [options] <directoryname or filename>

There are many options to the `ls` command, to modify what is displayed and how it is displayed. Without an option and argument, it will display just the contents of the current directory. Options can be combined together.
The `ls` command

```
$ ls
filea directory1 directory2
```

If a directory is listed as an argument, the CONTENTS of that directory will be displayed.

```
$ ls directory1
fileb
```

`-a` option displays any filenames that begin with a `.` character

```
$ ls -a
  .. directory1  directory2 filea
```

`-F` option places a `/` character after directory names

```
$ ls -F
directory1/ directory2/  filea
```

```
$ ls -aF
  .. directory1/ directory2/  filea
```
The *ls* command

-**R** option

This displays a listing of all contents of all directories and files under the specified directory.
The `ls` command

- `-l` option
This will display extended information about the contents, including:
Total blocks used in contents
Type, Permissions, Links, Owner, Group, Size (bytes), date/time created/modified
The *ls* command

$ ls -l
total 8
drwx------ 2 joeuser manager 4096 2011-08-24 13:31 directory1
drwx------ 2 joeuser manager 4096 2011-08-24 13:31 directory2
-rw------- 1 joeuser manager 0 2011-08-24 13:55 filea

Type: d for directory, - for regular file
Permissions: owner,group,other
Links: number of references to this object
   Directories have 2 because the . file refers to itself.
   For each subdirectory in a directory, this number increases by 1
Owner: used with permissions
Group: used with permissions
Size: in bytes
Date/time: When created or last modified
REMOVING A DIRECTORY

**rmdir command:** This command will erase a directory ONLY IF IT IS EMPTY.

**SYNTAX:** rmdir directory-pathname

**IF THE DIRECTORY IS NOT EMPTY,** use the **rm command with the –r option.**

**SYNTAX:** rm –r directory-pathname

Be careful, this will remove the directory and ALL of its contents, including any subdirectories in it!

**REMOVAL** of a directory and/or file is unrecoverable, unless a backup of the data exists elsewhere.
RENAMING/MOVING A DIRECTORY

mv command: This can be used to rename OR relocate a directory (or file)

SYNTAX: mv sourcedirectory targetdirectory
This command will work differently depending on the target directory name.
RENAMEING/MOVING A DIRECTORY

If the target directory already exists, the source directory will be MOVED into the target directory.

$ ls
directory1 directory2
$ mv directory1 directory2
$ ls
directory1
$ ls directory1
directory2
If the target directory doesn’t already exist, the source directory will be RENAMED to the target directory.

$ ls
directory1

$ mv directory1 directory2

$ ls
directory2
COPYING A DIRECTORY

**cp command:** This can be used to make a copy of a directory or file.

**SYNTAX:** `cp –r sourcedirectory targetdirectory`
This command will work differently depending on the target directory name.
COPYING A DIRECTORY

If the target directory doesn’t exist, a new directory with the contents of the source directory will be created.

```
$ ls
directory1
$ ls directory1
filea

$ cp –r directory1 directory2

$ ls
directory1 directory2
$ ls directory2
$ ls directory2
filea
```
COPYING A DIRECTORY

If the target directory exists, the source directory will be placed INSIDE of the target directory.

$ ls
directory1 directory2
$ ls directory1
filea

$ cp –r directory1 directory2
$ ls
directory1 directory2
directory1
$ ls directory2
directory1
$ ls directory2/directory1
filea
MANIPULATING FILES

Files are the core objects in a linux system. What you can do to a file is based on who it is owned by and the permissions on the file.

If you own the file (the default if you create it), you can do almost anything you’d like to it.

Files are created in many different ways:

– Text editing
– Compilation
– Output redirection
– Command execution
LISTING A FILE

ls command: This command will display information about a file

-1 option
This will display extended information about the contents, including:

Total blocks used in contents
Type, Permissions, Links, Owner, Group, Size(bytes), date/time created/modified
The *ls* command

```
$ ls -l
total 8
drwx------ 2 joeuser manager 4096 2011-08-24 13:31 directory1
drwx------ 2 joeuser manager 4096 2011-08-24 13:31 directory2
-rw------- 1 joeuser manager 0 2011-08-24 13:55 filea
```

Type: d for directory, - for regular file
Permissions: owner, group, other
Links: number of references to this object
(for files, this is 1 unless there is a link to this file)
Owner: used with permissions
Group: used with permissions
Size: in bytes
Date/time: When created or last modified
COPYING A FILE

cp command: This can be used to make a copy of a file. When you copy a file, the new copy is independent of the former one. Changes made to a file are not duplicated to its copies.

A file can be copied if the user has READ access to it. The file does not have to be OWNED by the user for it to be copied. The sourcefile and targets can be relative or absolute pathnames.

When new files are created with the CP command, the new file will be ‘owned’ by the user running the command (shown with ls –l)

SYNTAX: cp sourcefile target
This command will work differently depending on the target name.
COPYING A FILE

If the target is a FILE that ALREADY EXISTS, the target will be OVERWRITTEN with the source file. Normally you will be prompted before this occurs.

If the target is a FILE that DOESN’T EXIST, a new file will be created.

```
$ ls
directory1 filea

$ cp filea fileb
$ ls
$ ls
directory1 filea fileb
```
COPYING A FILE

If the target is a directory, the file will be COPIED into that directory.

```bash
$ ls
directory1 filea fileb

$ cp filea directory1
$ ls
directory1 filea fileb
$ ls directory1
filea
```
mv command: This command can be used to either rename a file, move it to an existing directory or do both at the same time.

SYNTAX: mv sourcefile target
This command will work differently depending on the target name.
RENAMING / MOVING A FILE

• If the target file exists, the TARGET file contents will be replaced with the SOURCEFILE contents. This is how you REPLACE a file.

• $ ls
• filea fileb
• $ mv filea fileb
• $ ls
• fileb
RENAMING / MOVING A FILE

If the target file doesn’t exist, the SOURCEFILE will be renamed. This is how you RENAME a file.

```
$ ls
filea
$ mv filea fileb
$ ls
fileb
```
RENAMING / MOVING A FILE

If the TARGET is a directory that you have WRITE access into, this will move the file from its current location to the existing directory.

```bash
$ ls
directory1 filea
$ mv filea directory1
$ ls
directory1
$ ls directory1
filea
```
**REMOVING A FILE**

*rm command:* This command is used for deleting a file.

**SYNTAX: rm filename**

The filename can be a path to a file in another directory. To remove a file you MUST have write permission on the directory it is in.

If somebody else owns the file you can remove it if you have write permissions on the directory it exists in.

Wildcards can be used to delete multiple files at the same time, along with listing multiple files after the `rm` command.
LINKING A FILE

In command: This command can be used to create another reference to an existing file OR directory.

Changes made to either the original or

SYNTAX: ln [-s] source destination

The file you are linking TO can be in any directory you have access to, and the destination can be in any directory you have write access in.

Links can be removed with the rm command.
There are two types of links: **HARD LINK** and **SOFT LINK**

**Hard link:** A hard link will continue to work even if you delete the original file that was linked to. Once both of those files are deleted, the data will no longer be available. This is the default when creating a link, when –s is not used. **You cannot create a hard link for directories.**

```
$ ls
filea fileb
$ cat filea
contents of filea here
$ ln filea filec
$ ls
filea fileb filec
$ cat filec
contents of filea here
$ rm filea
$ cat filec
contents of filea here
```
LINKING A FILE

Soft link: A soft link is used for creating links to directories AND files. When the original file is deleted, links WILL NO LONGER BE VALID.

```bash
$ ls
filea
$ cat filea
contents of filea here
$ ln -s filea filec
$ ls -l
total 0
-rw------- 1 user manager 0 2011-08-30 11:09 filea
-rw------- 1 user manager 0 2011-08-30 11:09 fileb
lrwxrwxrwx 1 user manager 5 2011-08-30 11:09 filec -> filea
$ cat filec
contents of filea here
$ rm filea
$ cat filec
cat: filec: No such file or directory
$ ls -l
total 0
-rw------- 1 user manager 0 2011-08-30 11:09 fileb
lrwxrwxrwx 1 user manager 5 2011-08-30 11:09 filec -> filea
```
LINKING A DIRECTORY

TO LINK DIRECTORIES:

`ln -s sourcedirectory targetdirectory`

$ ls
dira
$ ln -s dira dirb
$ ls -l
  total 4
drwx------- 2 user manager 4096 2011-08-30 11:12 dira
lrwxrwxrwxrwx 1 user manager  4 2011-08-30 11:12 dirb -> dira
more command: This command can be used to display the contents of a TEXT file on the screen. If it is used to display a non text file (binary, object file, etc) the display will be unreadable AND your session may freeze.

When used, this command will show one page at a time of the file, allowing you to press the space bar to show the next page.

SYNTAX: more filename

After each page is displayed, the output will pause and allow you to enter a single letter command:

    Spacebar – forward one screen
    b – backward one screen
    q – quit viewing the file
VIEWING TEXT FILES

SYNTAX: cat filename1

The cat command can be useful when merging files together.

SYNTAX: cat filename1 filename2 > filename3

This will create a new file ‘filename3’ with the contents being what is in filename1 and filename2.
**SEARCHING IN A TEXT FILE**

**grep command:** This can be used to find occurrences of particular text string inside of a text file. The results will be displayed on the screen.

**SYNTAX:** $\text{grep} \ [\text{-i} \text{ –v}] \ \text{string} \ \text{filename1} \ \text{filename2}$

By default, the string being searched for will be treated in a case sensitive manner (i.e. ‘dog’ will not be reported as found if Dog is in the file).

The **–i option** can be used to have grep ignore the case of the string.  
The **–v option** will display all the entries that DON’T match the string.  
The **string** can have spaces in it if it is put into single quotes.

**grep ‘red dog’ filename**  
Searches for the occurrences of **red** followed by a space and then **dog**.

**grep red dog filename**  
This tells grep to search for the string **red** in the files **dog** and **filename**.
PERMISSIONS / SECURITY

Unlike a single user system like Windows, linux is a multiuser system in which many users can be logged on at one time. To ensure that every users' data remains private, linux implements 

*discretionary access control* on files and directories.

Permissions are comprised of a combination of 3 privileges:

- **READ**
- **WRITE**
- **EXECUTE**
Creating / Editing Files

When writing a computer program, a text editor is used to create the source code file(s).

There are numerous text editors for Linux, with the most popular being:

- **VI** (pronounced Vee Eye)
- **EMACS**
- **NANO**
- **PICO**
EDITING TEXT FILES

Regardless of which editor you are using, remember to save your source code files into the proper subdirectories to ensure that your home directory is not one large folder full of files. Create a subdirectory for each assignment, preferably under a folder with the name of your course.

i.e.  ~username/computing1/program1
   ~username/computing1/program1/file.c
   ~username/computing1/program2
   ~username/computing1/program2/file.c
EDITING TEXT FILES

When you start an editor, you can either indicate the name of the file you want to create/edit at the command line as an argument to the editor OR open the editor and save your file using the editor’s built-in save command.

i.e.  editor-command-name myprogram.c
EMACS TEXT EDITOR

EMACS was written as part of the GNU project. It is launched at the command line by simply typing the command \textit{emacs}.

It’s preferred to indicate a filename after the emacs command to indicate that you are creating that file or editing that existing file.
EMACS TEXT EDITOR

Once emacs starts, start typing to enter text.

• Save your file by pressing
  ctrl-x followed by ctrl-s

• Quit by typing
  ctrl-x followed by ctrl-c
  If file has been changed, it will ask if you wish
  save changes or not.

Searching on the internet for an emacs tutorial may
be the best option for learning it.
VI TEXT EDITOR

VI is an editor written as the ‘visual’ version of what used to be a line editor called ed (an editor where you didn’t see the entire file at once on the screen, only one line at a time).

Because of this, the commands and usage of VI is cryptic. It is an important editor to learn as regardless of which version of linux or unix you use, it will ALWAYS be on the system.
VI TEXT EDITOR

The VI editor is started with the `vi` command, where it is preferred to give the name of a file as an argument, to either create a new file or edit an existing one.

VI has at least 2 modes: INSERT and COMMAND.
When the editor starts, you are in COMMAND mode. This means that any key you type is a command to make VI do something.

Typing the \textit{i} key will put you into insert mode, where you can start typing.

Pressing the ESC key puts you back into command mode, where you can use arrow keys to move around the screen.

Other commands:
- \textit{i} = insert mode, typing occurs at the cursor
- \textit{a} = append mode, typing occurs after the
- \textit{O} = open a new line above the current one
- \textit{o} = open a new line below the current one
- \texttt{:wq} = save the file and quit
- \texttt{ZZ} = save the file and quit

\textbf{REMEMBER}, you must press the ESC key before typing a command

Searching on the internet for a VI tutorial may be the best option for learning it.
PICO and NANO

PICO and NANO are simple text editors that work much like emacs, but don’t have all the features as emacs.

These are not normally found on every type of linux and unix system.

Might be easier to learn than emacs or vi, but it is important to at least learn EMACS or VI WELL!
PERMISSIONS / SECURITY

Every file and directory on a system has a combination of these privileges.

Each privilege will allow a different action to take place, depending on whether the object is a file or a directory.
READ permission

FOR A FILE: This allows the user to see the contents of a file using programs like cat, more and text editors like vi and emacs.

FOR A DIRECTORY: This allows a user to list the contents of a directory using the `ls` command.
WRITE permission

FOR A FILE: This allows a user to change the contents of the file using text editors, etc.
WRITE permission

FOR A DIRECTORY: This allows a user to delete, remove and rename files IN that directory.

A directory is actually a file that contains 2 entries for each file in it:

- INODE
- FILENAME
DIRECTORY CONTENTS

INODES

Each file has a unique number assigned to it, called the inode. The inode contains information like permissions, owner, etc. This is a pointer to a table on the system that contains the addresses of the blocks on disk that make up a file.
INODES

The *ls –i (or ls –li)* command will display the inode associated with the file(s) in a directory.

```
ken@cs:~/test3$ ls -li
total 216
1425706 drwx------ 2 ken manager 4096 2011-09-06 14:50 dir1
 770436 lrwxrwxrwx 1 ken manager  4 2011-09-06 14:52 dir2 -> dir1
 770433 -rw------- 1 ken manager  1462 2011-09-06 14:50 filea
 770435 -rw------- 2 ken manager 101992 2011-09-06 14:52 fileb
 770435 -rw------- 2 ken manager 101992 2011-09-06 14:52 filec
```

NOTICE THAT fileb AND filec HAVE THE SAME INODE – WHY?
Therefore, when deleting, renaming, copying, changing files in a directory, you are really **JUST CHANGING THE TABLE ENTRIES IN THAT DIRECTORY ‘FILE’**.
EXECUTE permission

FOR A FILE: This allows a user to run the file as a program (i.e. executing the commands inside of the file).

• In linux, a 'program' is either a text file that contains shell commands (a shell script) or a binary compiled version of a piece of source code (C language, etc).

• A program CANNOT be executed by the shell unless the program file has the execute permission attached to it.
EXECUTE permission

FOR A DIRECTORY: This allows a user to:

– `cd` into that directory
– use the directory in a path
PERMISSIONS

Directory and File permissions work TOGETHER to allow and/or disallow access:

Examples

1. File has **write** access
   Directory the file is in **does not have write** access
   The user **CANNOT** remove the file, but the entire contents of the file can be removed.

2. File has **read** access – no **write** access.
   DIRECTORY has **write** access.
   The user **CAN** remove the file, but the file contents **CANNOT** be modified.
Owner, Group, Other

Each file or directory has a set of these 3 permission codes (read, write, execute) for three categories:

**Owner** – These permissions affect the way the user who owns the file may use it. The owner of a file can override permissions on the file. A user’s account owns a file when it is created.

**Group** – A linux group refers to an entity that users are organized into. A user is part of at least one group. These permissions determine what other members of the group the file belongs to can do to it.

**Other** – This refers to the permissions for any user not in the group the object belongs to and that isn't the object owner.
PERMISSIONS

Each permission type is assigned a letter code:
  Read = r
  Write = w
  Execute = x

Each directory or file therefore has a combination of r, w, x for these 3 types of categories (owner, group, other). The permissions are displayed in this format:

<table>
<thead>
<tr>
<th></th>
<th>Owner</th>
<th>Group</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>rwx</td>
<td>rwx</td>
<td>rwx</td>
<td></td>
</tr>
</tbody>
</table>
PERMISSIONS

EXAMPLES:

```
rw-   r--   ---
```

Owner  Group  Other
In this case, owner has read/write, group has read and other has no permission.

```
 rwx   r-x   --x
```

Owner  Group  Other
In this case, owner has read/write/execute, group has read/execute and other has just execute.
MOST SECURE PERMISSIONS

The most secure permissions are those in which only the owner has access to the files and directories.

• Files: rw- --- --- (or rwx --- ---)
• Directories: rwx --- ---
VIEWING PERMISSIONS

The `ls -l` command is used to view file permissions.
`ls -ld` displays directory permissions.

EXAMPLE:

$ ls -l filea
-rw-r-----  1 user  manager  0 Sep  1 15:46 filea

$ ls -ld dir1
drwx------  2 user  manager 4096 Sep  1 15:49 dir1
The OWNER of a file can change the permissions on a file or directory using the `chmod` command.

**Syntax:** `chmod` permissions filename

Changing every item in a directory, including items in subdirectories can be done with the `-R` option for chmod.

`chmod -R permissions directory`a

2 methods of expressing permissions:

1. Add to, subtract from or replace existing permissions
2. REPLACE existing permissions
ADD TO, SUBTRACT FROM OR REPLACING EXISTING PERMISSIONS

The permissions argument specifies the permissions for user, group, other in 1 string:

user permissions, group permissions, other permissions

(u for user, g for group, o for other)

Not all of them need to be specified

For each listed, specify

+ followed by permissions to ADD TO EXISTING permissions
- followed by permissions to SUBTRACT FROM EXISTING permissions
ADD TO, SUBTRACT FROM OR REPLACING EXISTING PERMISSIONS

EXAMPLE:
chmod u+x,g+x,o-x filename

ADDS execute for user/group and DELETES execute for other

If doing the same operation on the same category, list the categories together:

chmod ug+x,o-x filename
REPLACING EXISTING PERMISSIONS

REPLACE existing permissions:

Each permission is assigned a numeric value:

- READ = 4
- WRITE = 2
- EXECUTE = 1

Add up the values to REPLACE the existing permissions and specify that value for each category (user, group, other)

(i.e. rwxr-x—x would be represented with 751)
PERMISSIONS EXAMPLE

chmod 755 filea

Sets permissions to

rw+xr-xr-x

chmod 640 filea

Sets permissions to

rw-r--r--
SAVING COMMAND LINE SESSION OUTPUT

The **script** command can be used to save the output of a command line session to a text file. Can be very useful when debugging programming compilation errors.

**Syntax: script filename**

The file specified will be overwritten with whatever is typed and outputted during the proceeding shell session. The session can be terminated with the CTRL-D sequence. SCRIPT output files can be treated as other text files are, i.e. they can be printed, emailed, viewed.
SCRIPT EXAMPLE

$ script outfile
Script started, file is outfile
$ ls
file1 file2 dira
$ ps -f
UID   PID  PPID  C   STIME TTY     TIME CMD
user  7691  7690   0 16:54 pts/4 00:00:00 bash -i
user  7694  7691   0 16:55 pts/4 00:00:00 ps -f
$ <CTRL-D>
Script done, file is outfile

$ cat outfile
Script started on Thu 01 Sep 2011 04:54:48 PM EDT
$ ps -f
UID   PID  PPID  C   STIME TTY     TIME CMD
ken   7691  7690   0 16:54 pts/4 00:00:00 bash -i
ken   7694  7691   0 16:55 pts/4 00:00:00 ps -f
$ exit

Script done on Thu 01 Sep 2011 04:55:45 PM EDT
**USING WILDCARDS IN THE SHELL**

When using commands which take filenames as arguments, WILDCARDS can be used to specify multiple files at once.

* - this matches all alphanumeric characters

[] - any 1 alphanumeric character inside of the brackets is matched

```
[abc] = a OR b OR c
[a-z] = any character from a to z
[A-z] = any character from A to z
```

Multiple subsequent patterns can be specified with the ',' character

```
[a-d,12] = matches a1 b1 c1 d1 a2 b2 c2 d2
```
WILDCARD EXAMPLES

ls a*

lists any files starting with a

more b*.c

displays any file starting with b and ending in .c

cat [a1].c

displays files a.c and 1.c

ls [a-z]*.out

lists any file starting with a-z and ending with .out
FINDING FILES AND DIRECTORIES

The *find* command can be used in order to locate files and directories in the filesystem.

**SYNTAX:**

```
find starting-directory -name pattern -print
```

*starting-directory* is where the search should start from, a full or relative pathname (can be just ‘.’)

*pattern* is the entry you are searching for. Wildcards can be used to locate objects containing a particular pattern. The pattern must be between single quotes (‘’) if wildcards are used

*-print* tells find to display the pathnames of the objects found onto the screen.
FINDING FILES AND DIRECTORIES

FIND EXAMPLES

find . –name myprogram.c –print

find / -name ‘*prog*’ –print
  finds any object with the string prog somewhere in the filename

find ~ -name ‘*.c’ –print
  finds any file that ends in .c UNDER the current user’s home directory
By default, your CS linux account has an email address associated with it:  

username@cs.uml.edu

When your account was created, the .forward file in your home directory was auto populated with your student.uml.edu email address.

Because of this, all email sent to username@cs.uml.edu will be instead sent to your student.uml.edu address.
CS EMAIL

To change where your CS Email is forwarded to, log into the cs.uml.edu server (or from a terminal in the linux workstation lab), use a text editor (vi / emacs) and change the email address inside of the .forward file to whichever address you would like.

IMPORTANT CS department related messages go to this email address, so it’s important to ensure that your cs email address does get forwarded!
PERSONAL CS WEB SPACE

Each CS account has built into it a personal web space. The **public_html** directory under your CS account is where you can put html code and related web files.

Your website address is

http://www.cs.uml.edu/~username

To ‘enable’ the site, you must first change the permissions of the **public_html** directory to be readable by ‘others’:

    chmod 755 ~/public_html
By default, any file or directory you create is accessible only by your account. This is important as it keeps your files secure from others. In order to a web site to work, the files and directories under public_html have to be readable (and traversable) by the ‘other’ permission category. To ensure these permissions are in place, you must do this for each file you create under public_html:

    chmod 604 filename

And you must do this for each directory created under public_html:

    chmod 705 directoryname

OR change everything at once in your website with:

    chmod -R g-rwx,o+rx ~/public_html