Sahil Gandhi

CS 35L

Office Hours: 9-11 AM 2432 Boelter Hall (Tuesdays)

**9/26/2016**

Cs.ucla.edu/~thuyvu/cs35L -> these are where the PowerPoint slides are.

* [thuyvu@cs.ucla.edu](mailto:thuyvu@cs.ucla.edu)

Final Exam: December 07, 2016 @ 11:30 AM to 2:30 PM

* One question will be about Parallel/Multithread

Brief Introduction of Linux:

* Kernel manages the system resources and communications
* Shells are the user interface of the kernel
* Applications are the things on top to
  + Things are either a file or a process

UNIX Basics:

1. You now have absolute paths. No more C: … everything is ‘/’
2. > my\_file = (over)write stdout to my\_file
3. >> my\_file = append to stdout to my\_file
4. < my\_file = content of my\_file is stdin
5. 0> my\_stdin, 1>my\_stdout, and 2> my)stderr
   1. 2> &1

Unic Commands

1. Man = manual
   1. To find something in the manual, use: /wordToFind
2. Cd to move (change directory), pwd = path to work directory
3. Which ls will tell you which ls is actually being run (there may be multiple instances of that installed through the system)
4. ~ = home, . = current folder, .. = parent folder, / = root/separator
5. History will tell you the commands that you have typed in the past.
6. grep = just searching for things
   1. history | grep ssh will only show the ssh things within history

Emacs:

1. To exit emacs, it is ctrl+x ctrl+c
2. Just use the emacs cheat sheet to work around and solve the questions.

**9/28/2016**

You can use touch to open a file -> touch a.file will open it up

* You can also use touch to change the timestamp it was created
  + –t 2100003103 –change timestamp
* Use the ln source target command to make links between files
  + Hard link = points to the same inode
  + Soft link = symbolic link … only contains the path to the linked file
* You would do soft link so you don’t have to always make new copies of programs or files in different places.
  + Think pointer to file name vs pointer to same inode
  + If you change the name of the original file, the hard link stays but the soft link would fail (or changing the path would fail the soft link)
    - Modifying a hard link or the original modifies the original
    - Hard links are NOT allowed to cross file systems because Inodes are unique to each file system

The rm command is used to remove the file, or directories or etc.

Find /usr/bin –name ? -> this is to find single lettered programs

Pipeline is the vertical bar | which allows one way communication between 2 processes

* This means program\_1 | program\_2 | program\_3
  + 1’s output will be 2’s input and 2’s out will be 3’s input
* Ls | wc – l will basically make the output of ls and try to do the word count of that
* Cat assign1.html | grep HTML
  + Searches assign1.html for the word HTML
* If you want to echo the output of the execution, all you need is echo ` \_\_\_\_\_ ` which is a back-tick, not single quote

Absolute path starts from the root, while relative path starts from the current folder

Sort will sort lines of stdin but it depends on the locale settings

Comm compares 2 sorted files line-by-line but also depends on locale settings

* Tr will be used to translate and delete characters

**10/3/2016**

Pipeline programs -> output of the first thing becomes the input of the next command

ProgramName \_\_symbol\_\_ fileName

* > will over(write) the output to the file
* >> will just append the output (NOT overwrite)
* < will use the file contents as stdin for program
* 0> stdin, 1> is stdout, 2> is standard error
  + 0> = <, 1> = >
  + You would use 2> if you need to run the same script millions of times
    - Output that to a file, but you only want the errors, so w.e is outputted to stderror will be written to the file
  + 2>&1 will make the standard error output to standard output

Comm is used to compare 2 files

Tr is used to translate or delete characters

* -d to delete. –s is to replace sequence of repeated chars into single occurrence

Sed is just a stream editor that can edit a program and pass it to an output

* Sed “s/regexp/replacement/g”
  + The /g makes it global

Wget link

Cat file\_name | sed “s/^ \*//g” newFile

* -> clean up empty strings
* ^ = not current, but next occurrence

Sed s clean out the tr’s

Grep for “<td>”

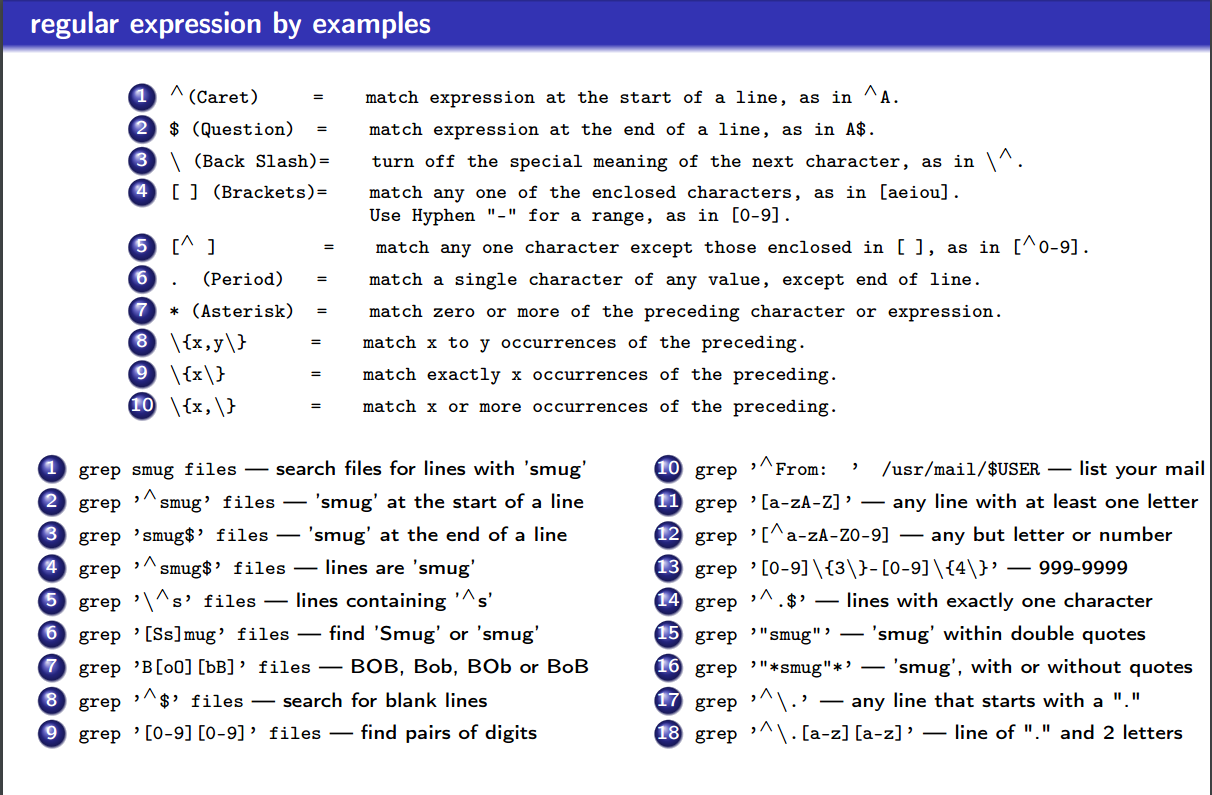
* Then sed “s/<td>//g”
* Grep for non-valid letters like “r”
* Sed “/[pk\mnwlhaceiou]/d”
  + Only contains these characters
    - Delete the rest that do not contain this
* Tr “[:upper:]” “[:lower:]” replaces upper case with lower case

Grep td, sed “s<td>//g”, sed “s\_</td>\_\_g”, tr “[:upper:]” “:[lower:”, sed “/^\s\*$/d” sed “s\s//g”, sed “/[^pk’mnwlhaeiou ]/d”

Shell scripting:

`

**Lecture #4 Notes 10/5/2016**



$1 will look at the first passed in argument

Find . –type f –maxdepth 1

Comm file\_1 file\_b -3 will check if 2 files are equal or not

GCC Compilation process -> Src (\*.c) and Header (\*.h) will go into the pre-processor (the .cpp file)

* Then it gets pre-processed (\*.i), then it goes through compilation (gcc)
* After, it goes through assembly (\*.s) and then to more assembly (as)
* Then the object is created (\*.o) and the static library is linked (ld). Then that outputs out the binary file that the machine can read ☺

**Lecture # 5 10/10/2016**

Updating C/C++ code:

On a \*.cpp file, you just recompile the changed code

On a \*.h file, it recompiles all those that include the header

make process

* Configure: check dependencies
* Make: compile and make executables (requires a Makefile)
* Make install: deploy or copy executables to the folder

Make file syntax:

Target: dependencies

System command

Example:

All: hello

Hello: main.o factorial.o hello.o

* g++ main.o factorial.o hello.o –o hello

main.o: main.cpp

* g++ -c main.cpp

factorial.o: factorial.cpp

* g++ -c factorial.cpp

hello.c: hello.cpp

* g++ -c hello.cpp

clean:

* rm \*o hello

CC can be anything you want

CC=g++ or w.e -> can

If the timestamp of a file is after the make file, then the makefile will ignore it.

If you write makeinstall again, it will overwrite any new changes that you made

You need to copy it from the archived link

Copy from the NEWS | 5++ to the very end

Use patch –p1 < mypatch

Then hit make

Python basics:

Duck-typing: no matter what u type, the variable is automatically declared into the appropriate variable

Indentation matters … no braces

String has different concatenations

List is dynamic and heterogenous (you can mix and match here)  
 comment using # or “’ ‘”

Tuple is like an array but with the same time

Set is a container

Dict is for dictionary

Control flow is similar to C++ or Java

Functions also exist in

1 main: <https://docs.python.org/2/library/__main__.html>

For assignment 3, we are given randline and then we need to fix randline to do things

* have –u option to

Prime sleeve -> generate a list of prime numbers

* learn how to optimize operations and such

**Lecture # 6 10/12/2016**

Download randline.py to be able to run it ->

* then run python3 by using

/usr/local/cs/bin/python3 randline.py … etc

Use the sorted command to sort through the files -> then compare that to unsorted to see if you need to sort it at all or not.

**Lecture #7 10/17/2016**

If you just add an extra enter in the source code, the path might become invalid

* you need to make a copy of the source code that you have so that your friend knows what changes have been made
  + Use GIT!

Centralized version control:

* 1 central repository, changes in one place
  + Subversion
* Very simple, changes are immediately seen. There are NO backups (BAD)!

Distributed Version Control:

* Multiple repositories,
* Push and commit to your repo. Parent repo will pull and update
  + Eg. Git, bazaar, bitkeeper
* Can do it offline, and very fast
* BUT you need lots of space and download time

Repository – files and folders of a full history and versions (database of changes)

Working copy – a local copy of a local repo

Check-out = making a local working copy from repo

Commit = writing changes made in working copy to repo

Git objects (4 types)

1. Blob (binary large object) -> content of a file (w/o name, timestamp, etc.)
2. Tree -> a folder with filenames, each one is (name+blob) / a tree / soft-link
3. Commit -> points to the top-level tree of the git commit –d project
4. Tag -> name to a commit object

The way that git works -> a file can be in one of three states

1. Committed state = it is safely stored in the local repo
2. Staged = file is marked to be committed in the next commit snapshot
3. Modified = it is changed by you but not yet committed

Git init to create new repo

Git clone to copy a repo to your local machine

Git add will add files to index

Git commit will add changes to repo

Git help, git status, git log, git show, git diff will show different things related to the file

Git checkout will allow you to check out a specific version/commit

Git reverse will let you reverse a commit using a patch

**Lecture # 11 10/31/2016**

Kernel is the core of the OS -> interface between hardware and software

* Controls access to system resources: memory, I/O, CPU
* Ensures protection and fair allocation

User space -> where normal user processes run

* Limited access to system resources (memory, IO and CPU)

Kernel Space -> stores the code of the kernel, which manages processes

* Prevents processes from messing with each other and the machine
* Only the kernel code is trusted

User space (user apps + GNUc lib) -> Kernel Space = (syscall interface, kernel, dev drivers) -> Hardware

System calls: A part of the kernel accessible from user space (applications)

* The executable is interrupted and its control is passed to the kernel to perform if valid

Function calls vs System calls:

* Function calls have a caller and a callee are of the same process, same “domain of trust”
  + Getchar, puthcar, fopen, fclose
* System calls transfer control form untrusted user process to trusted OS
  + Thus more expensive performance wise
  + Read, write, open close

Five steps for system calls:

1. Interrupt process and save state
2. OS takes control of CPU and checks if valid
3. Perform requested action
4. Restore the state and switch back to user mode
5. Give CPU control back to process

What can go wrong with system calls -> internally, efficiently (minimizing system calls) … like cat stdin

* Ex. Char buffer[1];
* While (read(0, bugger, 1) > 0)
  + Write(1, buffer,1)

If you delete files using rm … you can do

Cd .snapshot -> which is a folder with the name snapshot ... that will contain the files that you just deleted!!

**Lecture Notes 11/7/2016**

Parallelism is used to execute several computations at the same time and comes in two ways:

1. Multi-tasking where several processes are scheduled alternatively or possibly simultaneously on multiprocessing systems
2. Multithreading is where job is broken down into pieces and threads take up each of the pieces and execute it simultaneously

Process – each process has its own address space

In tr ‘A-Z’ ‘a-z’ | sort | comm -23 –f

* Process 1 = tr, Process 2 = sort, Process 3 = comm
* Thread = flow of instructions in a process
  + Smallest unit of processing scheduled by the OS
  + A process = a main thread + other threads
* One processor will switch between different threads (pseudo parallelism)
* Multiprocessor -> threads run on different cores (true parallelism)

ALL THREADS: share same address space, but private stacks

Shared memory in threads:

* Powerful -> easily access and share among threads
* Efficient -> easy to spawn and kill and no need for system calls when sharing data
* Non-trivial -> race conditions can happen and you need synchronization
  + Access AND edit a variable

Thread APIs that we need to include:

* Pthread\_Create -> creates a new thread within the process
  + Use this to just make new threads
* Pthread\_join -> waits for another thread to terminate
  + When you want a thread to finish first before you want to do some new command
  + You don’t actually know WHEN or how long a thread will take so you need to pjoin to avoid race conditions
* Pthread\_equal -> see if the thread ids are equal
* Pthread\_self -> returns the IDs of the calling thread
* Pthread\_exit -> terminates the currently running thread

You may not know the order of when threads are connected (even if you create them in order according to the code) … THAT is why the join command is so powerful and so much required!

To create a thread:

Int pthread\_create (id, attr, routine, arg)

* Pthread\_t \*restrict thread, const pthread\_attr\_t \*restrict attr, void \*(\*start\_routine)(void\*), void \*restrict arg)
* This will create a new thread with handle thread and attribute attr. This will execute the start\_routine with arg as its sole argument
  + Attr is just the different attributes (sub requirements for the thread)
  + Restrict means:

Int pthread\_join (pthread\_t thread, void \*\*value\_ptr)

Why do we have the same numbers print or less than 10 threads running -> its possible a thread from before is NOT done

Gcc test.c –o test –pthread

**Lecture notes 11/9/2016**

17296 is number of lines that we are trying to make!

Multithreading is not supported yet … so we want to be able to do that

* As long as you write make check, and the output is the same … then you get a hundred.
* Either you get it or you don’t get this homework!

**Lecture notes 11/14/2016**

Presentation for Week 10:

30 minutes to read roughly, 3 hours to write draft/research, 5 slides (problem story, etc … not counting title slide and questions slide)

* Report = 600-1000 words.
* Cite sources -> using \_\_\_\_ Format

Libraries are used to package similar functionality -> modular programming

* Linux supports 2 types, static and dynamic

Static: Functionality is to bind to a program statically at compile time

* \*.a files ->

Shared libraries are linked dynamically or loaded dynamically

* Dynamic linking = have linux load library upon execution
* Dynamic loading = selectively calling functions with the library in process
  + Happens at run time
  + YOU are in control of what is being loaded
  + For example, you might be asking for the user to input something, and based on that it will only load the libraries/functions related
* Another example … lets say 1000 functions in a file (each using different libraries) -> wait till user says which functions to run before just linking everything!

**Lecture Notes 11/16/2016**

SSH secure shell -. Verify host and uses public key in ~/.ssh/known\_hosts

* Future host validation uses this public key to double check (if not match)!

Symmetric vs asymmetric keys:

* Symmetric = one key for both encrypt and decrypt
* Asymmetric = pubic key to encrypt and private key to decrypt
  + In pair!!

Client authentication: two authentication methods

* Password based: username and password
* Key based -> use public and private keys
  + Client generates both keys, and keeps the private key. Give public key!
  + Copy public key to ~/.ssh/authorized\_keys
  + Server authenticates if client shows to obtain private key
  + Single sided, only friend with public key can message you. If you want 2 way communication, you need

**Lecture 11/21/2016**

Things on the final:

1. No emacs!
2. Softlink vs hardlink
3. Go over his slides,
4. One question about Chmod and the various + ‘char’ questions
5. Why do you need to have locale?
   1. Different formats, languages, time zones
   2. Sort might have different outputs with diff locales (C vs LC\_ALL, etc)
6. Difference between interpreter and compiler!
   1. GCC is a compiler, but python is an interpreter -> shell is also an interpreter
7. Shell programming!
   1. Back tick executes the command
      1. Nfiles = `cat history`
   2. Single quote is literal meaning
   3. Double quote is to just
8. Regex and Sed will take up about 5-10% of the final!
9. Makefile will be on the final! -> look at week 3 and the week 8 ones for help
   1. Hello.o: hello.cpp ….
      1. G++ -c hello.cpp
   2. Clean:
      1. Rm \*o hello
10. Python -> will not be on there!
    1. But you need to learn about the python 3 differences vs python 2 differences
    2. Keywords, classes, text models, int division, things like that!
11. Git and git workflow will be useful as well!
    1. The picture on the slide (11/14) -> easy questions hopefully!
    2. Spend a few minutes to read the spec of assignment 4 and make sense of EVERY step you are asked to do!!
12. Functors can be used in C and C++
    1. Will be on the final!!!!
    2. Remember pointers vs addresses vs everything!
13. System calls -> need to be sure for that homework part of assignment 6, if you know the size of the buffer that you are going to read, and use the read command to read all of it, you will read it all of it at once (one system call)
    1. Pros and Cons of system calls vs Standard I/O
       1. Atleast 2 pros and atleast 2 cons
14. Multithreading will be on there a little
    1. He has a good question that will be put on there!
15. SSH – public vs private key
    1. When to use either one, and advantage/disadvantage

**Various topics will be on the final!!!!**

1. **Softlink vs hardlink**

Softlink is when you have the address to a particular file or folder or program

* Can be used across multiple file systems! But if name or path is changed, then you will get errors
  + Good for when you don’t want multiple copies of your program/files or if they are in different locations
  + Pointer to the name of the file essentially

Hardlink is a pointer to the same Inode of a particular file or program

* It is NOT allowed across file systems since inodes are unique to each file system
* BUT if you change the contents or anything, then it will remain the same since it is same inode
  + Modifying hardlink or the original modifies the original
  + Requires a lot of space unlike soft link

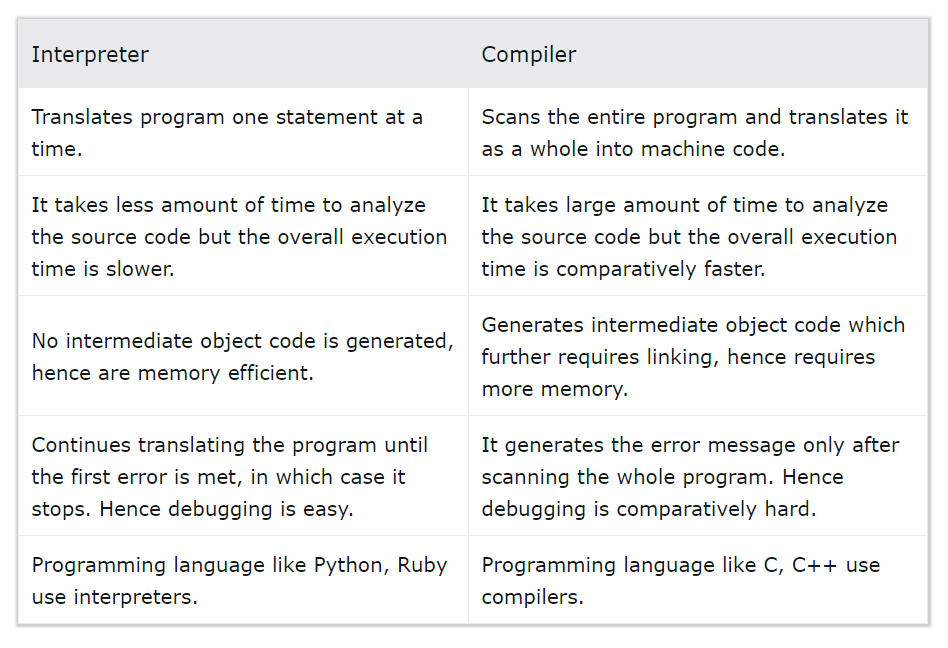
1. **One question about Chmod and the various + ‘char’ questions**

Chmod is used to give or take permission for particular files or programs

* Literally called change file mode bits
* + means to give permission, - means to take away permission
* Use –R to make it recursive
* x = execute permission, s = set user ID or group id on execution, r = read, w = write, X = execute/search only if user already has execute permission, t = restricted deletion flag
* Chmod can’t change symbolic links permissions
* A numeric mode is from one to four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. Omitted digits are assumed to be leading zeros. The first digit selects the set user ID (4) and set group ID (2) and restricted deletion or sticky (1) attributes. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group, with the same values; and the fourth for other users not in the file's group, with the same values.
* Ex: chmod u=rwx,g=rx, o=r myfile
  + U = user permis, g = group perm, o = other perm, a = all permissions
  + Chmod u+x = give user execute permission
* Chown is used to give permission to others -> chown –R user\_name \_file/FolderName\_ -> -R = recursive

1. **Why do you need to have locale?**
   1. Different formats, languages, time zones
   2. Sort might have different outputs with diff locales (C vs LC\_ALL, etc)

The purpose of locale is for different formats and languages to be outputted and displayed properly

* Especially important for sorting so that things are sorted with the correct conditions that you want (like capitals come before lower case or vice versa)
* LC\_ALL overwrites all locale parts

Else, you need to LANG=en\_GB.UTF-8

LANGUAGE=

LC\_CTYPE="en\_GB.UTF-8"

LC\_NUMERIC="en\_GB.UTF-8"

LC\_TIME="en\_GB.UTF-8"

LC\_COLLATE="en\_GB.UTF-8"

LC\_MONETARY="en\_GB.UTF-8"

LC\_MESSAGES="en\_GB.UTF-8"

LC\_PAPER="en\_GB.UTF-8"

LC\_NAME="en\_GB.UTF-8"

LC\_ADDRESS="en\_GB.UTF-8"

LC\_TELEPHONE="en\_GB.UTF-8"

LC\_MEASUREMENT="en\_GB.UTF-8"

LC\_IDENTIFICATION="en\_GB.UTF-8"

LC\_ALL=

1. **Difference between interpreter and compiler!**
   1. GCC is a compiler, but python is an interpreter -> shell is also an interpreter (in picture above)
2. **Shell programming!**
   1. Back tick executes the command
      1. Nfiles = `cat history`
   2. Single quote is literal meaning
   3. Double quote is to make sure that separator characters or anything else perform as they are supposed to and are not globbed together

In general always enclose everything in double quotes. The dollar sign means the actual literal variable

* !/bin/bash -> this is the “shebang” that lets us

A single bracket ([) usually actually calls a program named [; man test or man [ for more info. Example:

$ VARIABLE=abcdef

$ if [ $VARIABLE == abcdef ] ; then echo yes ; else echo no ; fi

yes

The double bracket ([[) does the same thing (basically) as a single bracket, but is a bash builtin.

$ VARIABLE=abcdef

$ if [[ $VARIABLE == 123456 ]] ; then echo yes ; else echo no ; fi

no

Parentheses (()) are used to create a subshell. For example:

$ pwd

/home/user

$ (cd /tmp; pwd)

/tmp

$ pwd

/home/user

As you can see, the subshell allowed you to perform operations without affecting the environment of the current shell.

4a. Braces ({}) are used to unambiguously identify variables. Example:

$ VARIABLE=abcdef

$ echo Variable: $VARIABLE

Variable: abcdef

$ echo Variable: $VARIABLE123456

Variable:

$ echo Variable: ${VARIABLE}123456

Variable: abcdef123456

4b. Braces are also used to execute a sequence of commands in the current shell context, e.g.

$ { date; top -b -n1 | head ; } >logfile

# 'date' and 'top' output are concatenated,

# could be useful sometimes to hunt for a top loader )

$ { date; make 2>&1; date; } | tee logfile

# now we can calculate the duration of a build from the logfile

There is a subtle syntactic difference with ( ), though (see [bash reference](http://www.gnu.org/software/bash/manual/bashref.html#Command-Grouping)) ; essentially, a semicolon ; after the last command within braces is a must, and the braces {, } **must** be surrounded by spaces.

1. **Regex and Sed will take up about 5-10% of the final!**

Grep vs Sed vs Awk:

Grep = mainly just finding things (can be from multiple lines) that match a certain pattern

Sed = can do the above BUT can also modify things such as replacing, deleting, or more

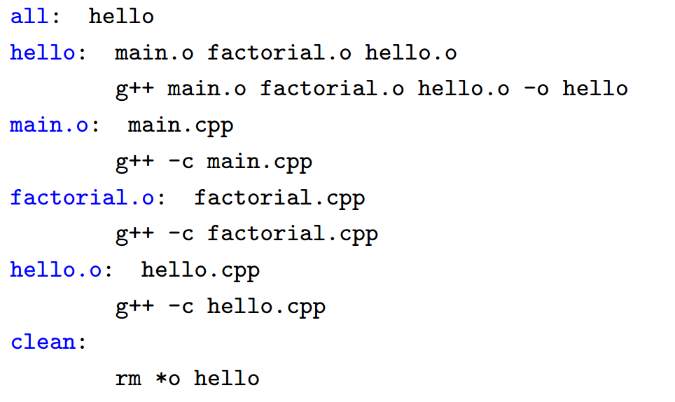
Awk = very similar to sed, but harder to use and mostly used for reading CSV style files, and processing records and printing

Sed’s –e command is technically optional unless you are trying to run multiple expressions, then separate each expression with a –e command preceding it -> sed –e ‘s/trump/johnson/’ –e ‘s/johnson/Bernie/’ myFile.txt

* To use regex with grep, just grep ‘regex’ -> like grep ‘<td>\(..\*\)<\/td>’ <&0 means take from the input, look for anything with td in the beginning and td at the end -> the \ are just used to escape (, ), and / respectively!
* Also use sed –r so that you can use extended regex … or you can use [:digit:] or something
* \g is global (all occurrences on EVERY line), \d means to delete, \p means to just print out certain things
  + Sed –e –n = use e and n (silent option)
  + $ sed -e '1,10s/enchantment/entrapment/g' myfile2.txt ->> replaces ONLY on lines 1 through 10
  + You can also use : to separate, like ‘s:\_\_\_:replace/g’ /// it works!
  + $ sed -e 's/.\*/ralph said: &/' origmsg.txt -> the &/ means for EVERY matched thing, precede with the ralph said: line … pretty useful right?
  + $ sed -e '1!G;h;$!d' forward.txt > backward.txt
    - The 1!G (get) tells it to apply to everything except 1st line, $!d is apply the delete to every line except the last line
    - H = put into hold space … will hold different things until last line will output (and then everything in the hold space) -> this is how to reverse a file!!
* exp !~ /[*regexp*](https://www.math.utah.edu/docs/info/gawk_22.html#GLOSS58)/ -> prints out if things matched not equal to the regexp

Common regex things:

* Anything in parenthesis will be grouped together like ^(file\_\w+)
* + is kleene operator = 1 or more repetitions, \* = Kleene star = 0 or more repetitions
* \w = alphanumeric, \W = nonAlphanumeric => can also do A-za-z0-9 for ranges of things
* \s is any whitespace, \S is any non-whitespace character -> ^ = starting from beginning to $ end (^…%+$)
  + Just by itself ^ means exclude (like [^abc] means don’t choose abc)
* Use ? after a character or group to make it optional(like (ab?C) means abC or aC)
* Use {a,b} to make it a range of selections, at min a times and at max b times OR {a} exactly a times
  + ^([\w\.]\*) which will match emails starting with alphanumeric characters including the period.
  + Can also use ^([\w|\.]\*) -> alpha numeric OR a period
  + [] is a list soo ^^ is matching a SINGLE character present in the list
  + () is a matching group … so only do that when you want a group to be matched/captured
  + ‘d\b’ means word boundary, so between 2 ds that are at the end of a word (and next char is non alpha)
* Example: (\w+)\.(jpg|png|gif)$ -> means match only which have alphanumeric character in beginning, then dot, then jpg or gif or png BUT from the back (the $ means back)
  + ^\s\*(.\*)\s\* -> skipping white space, capture all characters in between, and then skip whitespace after
  + (\w+)\(([\w\.]\*):(\d+)\) => watch alpha until ( which was escaped, then alpha or period, then number
  + (\w+)://([\w\-\.]+)(:(\d+))? -> match alpha till :// then alpha or . or – then a :digit is optional as third group
  + echo "(555)-555-5555" | sed -r 's/\(([0-9]{3})\)\-([0-9]{3})\-([0-9]{4})/\1\2\3/g'

1. **Makefile will be on the final! -> look at week 3 and the week 8 ones for help**
   1. Hello.o: hello.cpp ….
      1. G++ -c hello.cpp
   2. Clean:
      1. Rm \*o hello

Syntax =

target: dependencies (and then on next line, the system commands)

In general, for makefiles, if you have dynamic libraries, then you want to makefile look like the following:

CC = gcc

CFLAGS = $(OPTIMIZE) -g3 -Wall -Wextra -march=native -mtune=native –mrdrnd

randlibsw.so: randlibsw.c

$(CC) $(CFLAGS) randlibsw.c -shared -fPIC -o randlibsw.so

Similarly, to make object file, you have what is given in the solution, then in “hello”, you see that g++ with the –o option combines the object files into an executable

* Other syntax is gcc –o myExec.exe myObject.o // actually this seems to be preferred

1. **Python -> will not be on there!**
   1. But you need to learn about the python 3 differences vs python 2 differences
   2. Keywords, classes, text models, int division, things like that!

Duck typing language -> means essentially no types, if you pass in func doSomething(mallard) -> mallard.quack() will run the correct classes quack, no casting required

Printing: python 2 can have (‘---‘) or just ‘---‘, whereas python3 MUST have the parenthesis

Integer div: python2 do, 3/2 = 1 whereas in python 3, 3/2 = 1.5

Exceptions: Python2 had try: … except NameError, err or except IOError as (errno, strerror) … python 3 will only take the 2nd but python2 will take both, whereas python3 has try: … except NameError as err

For loops: python2, variables leaked out globally while in python3 it does not (ie i<5, i++, if you print after, then python2 will have 5 while python 2 would just have the initial value)

Rounding: Python2 rounds 0.5 up, but python3 just rounds 0.5 to the nearest even value

Str: python2 had str(), separate Unicode() but no byte type … python3 has unicode str, 2 byte classes (byte and bytearay)

* By default, str and byte are used (where str is using Unicode)

Classes: python 2 has classic up till 2.1 and new after 2.2 -> basically in classic, type(x) is <type ‘instance’> while in new type(x) is x.\_\_class\_\_ (both are technically instances of x though) -> python 3 basically has it always be new type

1. **Git and git workflow will be useful as well!**
   1. The picture on the slide (11/14) -> easy questions hopefully!
   2. Spend a few minutes to read the spec of assignment 4 and make sense of EVERY step you are asked to do!!

Git is distributed version control rather than centralized version control -> answer When, Who, Where, How, What, and Why

* You have git trees, commits, globs (the folders and programs and such), and git tags (these explain the commit number and person and other things about the commit … so basically associated with the commits)
  + Files are either committed (safely stored in local repo), modified (changed but not committed yet), or staged (marked to be committed in the next commit snapshot)
  + Git add puts it to staged, commit makes it back to unmodified (for the next commit), then you modify and commit, git rm will just put it in untracked (and also delete the file in question)
* If you have workspace, local repo, and remote repo, then you fetch from the local repo but then pull from the workspace
  + You can use git checkout \_branchYouWantToCheck –b \_nameofurnewBranch\_
  + git format-patch 62ca21c8c1a5aa3488589dcb191a4ef04ae9ed4f --stdout -1 > quote-patch.txt
    - Git commit –F // this is used for committing files
    - Git am -> apply a series of patches from a mailbox

1. **Functors can be used in C and C++**
   1. Will be on the final!!!!
   2. Remember pointers vs addresses vs everything!

void\* routine(void \* startPos) -> that is a functor … use int startingPos = \*(int\*) startPos; to translate that startPos into an int

* int frobcmp(const void\* chara, const void\* charb) // each \* points to an array of non-space bytes + space
  + char const \* wordOne = \*(char const \*\*) chara; // getting a single character pointer from the array of words
  + char const \* wordTwo = \*(char const \*\*) charb;
  + this is how to pass into qsort
  + qsort(strarr, arraySize, sizeof(char\*), frobcmp);
* In general: buffer is a pointer, so \*buffer = the object at buffer. B is a char … we want to put the pointer so &b is the address (pointers are just addresses!!)

int byte = getchar();

while (byte != EOF)

{

if (charMap[byte] == 0)

putchar(byte);

else

putchar(charMap[byte]);

byte = getchar();

}

char buffer[1];

while (read(0, buffer, 1)> 0)

{

int num = \*buffer;

char b = charMap[num];

if (b == 0)

write(1, buffer, 1);

else

write(1, &b, 1);

}

As you can see in this case we just putchar a char directly! As long as you use realloc instead of malloc to relocate a dynamic array, you should be able to traverse it properly using just ++ operator to move forward

* unsigned char \* currWord = (char \*) malloc(allocSize \* sizeof(unsigned char)); // pointer to the current word
* currWord = realloc(currWord, (currLength\*2)\*sizeof(unsigned char));
  + then you can just do unsigned char\* currChar = currWord; ++currChar to move forward!!

1. **System calls -> need to be sure for that homework part of assignment 6, if you know the size of the buffer that you are going to read, and use the read command to read all of it, you will read it all of it at once (one system call)**
   1. Pros and Cons of system calls vs Standard I/O
      1. Atleast 2 pros and atleast 2 cons

Pros:

* System calls are easy to implement and easier to use
* System call performance on Linux is very fast and highly optimized (especially if you know the size of the buffer that you want to use to read/write)
  + Else it is slower!!

Cons:

* You need a syscall number which needs to be assigned when you create the kernel …
* System call is overkill for simple exchanges of information
* System calls aren’t easy to use in scripts and can’t be accessed directly from filesystem
* Every architecture needs to be able to separately register and support the system calls
* System calls might expose underlying system dependency -> easier to hack??

The alternatives to implementing a syscall:

* Implement a device node and read() and write() to it. Use ioctl() to manipulate specific settings or retrieve specific information.

1. **Multithreading will be on there a little** 
   1. He has a good question that will be put on there!

Multi-threading != multi-tasking -> the second one is just splitting tasks in such a way that they are scheduled alternatively or broken into chunks and are handled interchangeably. The first (multithreading) is actually splitting the program into chunks (threads) which can then be executed simultaneously on a multi-threaded system

* On single processor, multi-threading will create pseudo parallelism …. The processor is actually switching between different threads
  + With multiple cores, each core will get a different thread … true parallelism -> threads share same address space but have their own private stacks (same address space means easily access and shared among threads and easy to spawn/kill (efficient) … no need for system calls) BUT you might have race condition!
* APIs that we need to know: pthread\_create and pthread\_join … basically you want to create threads and store their count (ID number) somewhere … that way after the main thing is done running, you execute the join so that you can get all the threads to a particular point before continuing on to the rest of the program
  + The “restrict” option means that the pointer to this function do not alias (aka they do not point to the same object)
    - We don’t really need it for pthread\_create perhaps for memcpy, it would actually be useful
* int createThread = pthread\_create(&threads[i], NULL, routine, (void\*)(startPos + i)); // 0 = no error found
  + Notice that first parameter is the value thread[i], second is NULL (we don’t care about restriction attributes), third is routine (this is a FUNCTOR!!), and fourth is the argument we want to pass to our functor
    - void\* routine(void \* startPos) // as you can see, a functor!!
      * int startingPosition = \*(int\*) startPos;
* pthread\_join(threads[i], NULL); // this is how to close them
  + The second which is the void\*\*value\_ptr is null since we don’t really need that.

1. **SSH – public vs private key**
   1. When to use either one, and advantage/disadvantage

In general for this, one person has the private key and then the public key is given out to some people -> the public key is used to decrypt the message … if it is readable then it must have been using the private key that I have and therefore the sender is me (atleast it was encrypted using my private key)

* Similarly, if people send me things encrypted with the public key, I can use my private key to decrypt it and if it isn’t garbage then I know that it was sent by someone to me specifically using my public key
* You would use a public key to give to anyone else, private key + passphrase to keep to yourself
* Advantages: Easy to use public/private key combo, and easily encrypt entire file systems from outsiders
* Disadvantages: Easy for someone to decrypt your data with a public key (maybe insecure method of transferring the key) -> use multiple keys and spread them out so that if someone is hacked, only their small portion of files is hacked

Signature basically is just used to verify that you were the last person to change/modify the file/program and that it is indeed yours

* It however does NOT do anything against if someone is able to intercept ur signature or your laptop … then no real
* way to verify that it was indeed someone else who wrote that code!
* Password easy to forget and brute force attack!

1. **Dynamic linking**

Dynamic linking is useful in several different cases: there are static and dynamic libraries ->

* Static means functionality (like external methods and such) are bound at compile time (.a files)
  + Slightly slower and takes up more space -> need to give EVERYTHING to make the program run
    - Therefore it might not be as useful to mass users unless you want your program files to be large
* Dynamic means that they are bound at run time
  + MUCH faster and bug fixes/upgrades can propagate through product without shipping more stuff
  + Reduces total resource consumption -> in the form of a .so
  + There is also dynamic loading which is selectively loading at run time which specific functions you want to load
    - We did this in project 8 -> we load shared libraries from disk (file) into memory and re-adjust the location …. All done by a library named ld-linux\*.so.2 -> load-time relocation and position independent code (PIC)
* Api calls that we need:
  + Void \* dl\_handle; dl\_handle = dlopen(“randlib.so”, RTLD\_LAZY) // theres also RTLD\_NOW which means all undefined symbols happen before dlopen returns. Lazy makes it so if symbol is not referenced, it is not resolved. Also RTLD\_GLOBAL and RTLD\_LOCAL, both of which say that symbols defined in this library are/are not made available to resolve references in subsequently loaded libraries
  + Void (\*initialize) void, initialize = dlsym(dl\_handle, “rand64”) // this is how to get a reference to the method … then we can just call initialize();
  + Then at the end, dlclose(dl\_handle); // this will close everything
    - We can also have dlerror (char\* error = dlerror();) which can tell us if there is an error
    - Also have \_\_attribute\_\_ ((\_\_constructor\_\_)) and destructor to run automatically when dlopen() and dlclose() are called respectively
* To compile, use the ta ppt -> gcc –shared –fPIC (position indep) greting-fr.c –o greeting-fr.so
  + Gcc –ldl –wl, -rpath=. Greeting-dl.c –o greet-dl