CS 35L Notes

Lab 1 9/26/19

* Computer is basically a machine that does various computations
* Kernel is the core of the OS, allocates time and memory to programs, and handles file system and communication between software and hardware
  + Lies in between the hardware (low level) and software (high level)
* Shell is the interface between the user and kernel, interprets commands and takes necessary actions to carry out commands
* Programs abstract away the shell, interact with the shell
* Daemon is a process that runs in the background
* Will be using Ubuntu in this class, use linux servers with SEASnet login
* Command line interface (CLI) is usually through a shell, lines only, while graphical user interface is an interface with visual elements
* CLI can be used with other programs, while GUI can only be used by humans
  + For CLI, you can send in parameters with starting the program
* A GUI takes more resources
* A file is a collection of data and a process is something executing the data in a file
* In Linux, everything is a process or file, and each one has a PID (process ID)
* Files are a collection of data: document, executable, directory (counts as file too!), and devices
* Linux files are structured in a tree, where there’s a root and then each subfolder is another level
* Absolute path is path to the root and relative path is path from the current working directory
* For relative directory, go back up to the first common shared directory, and then go down to the desired file
* If you see the ~, then it means you’re in the home directory
* Pwd for print working directory
* Cd for changing directory
* Clear for clear screen
* For cd .., can chain it together with cd ../../.., etc.
* A single dot means current directory (cd . does nothing)
* Cd ~ moves you to the home directory
* Can use cd with either relative or abosolute path, and then cd returns you to the previous
* Use “man” for manual
* Ls, use ls –l for more options
* When you use ls, it diesplays file names in ASCII code order (alphabetical)
* Different files are colored differently (green is executable), dierctories are marked with a d, owner of the user is lited, then usergroup that user belongs in, then file size
* 8 bits makes 1 byte, 1024 bytes make up one kilobyte, 1024 KB makes one MB, etc.
* To search for something, then n for next, or shift-n for previous
* Ls –al shows the hidden files (files with a . in front)
* Ls –al is the same as ls –a –l
* The second number is the number of subdirectories plus 2
* Rwx means read, write, and execute permissions
* Whoami
* Which ls gives the alias of ls
* To make a file, use touch
* Cat shows the contents of the given file, and also concatenates them together
* Use tab for autocomplete
* Echo also prints stuff onto the terminal
* Cp for copy, need to specifiy source and target
* Mv moves files, need to specify source and target as well
  + Source file disappears for mv but not for cp
  + Like cut versus copy
* Ln is used to create a link, either hard link (point to physical data), or soft link (symbolic link, -s, points to a file)
* Touch *filename* changes the file’s timestamp: touch –t 201901031759.30 *filename* changes file’s timestamp
* Symbolic link points to a file, if you delete the file, you get a dangling symlink
* Hard link points to the inode (data) of a file, and deleting the file does not affect the link
  + Can touch the underlying data, so it is a hard lik
* An inode stores the metadata and disk block locations of the file object
* Directory is just a glorified file, contains links to inodes, which then point to data blocks
* Chmod changes mode, there’s read, write, and executable
* There’s three groups: u (user), g (users part of file group), o (others, users not owner of file or members of group), and a (all three, aka ugo)
* Chmod: use with + to add specified modes, - to remove specified modes, and = to specify exact modes
* Chmod changes the rwx permissions
* 3 bits, so up to 8 states of permissions
* Chmod ug+rw mydir does user and group, add read write permission
* Chmod a-w prevents all from writing it
* wh commands does various searches, whatis<command> returns Name section of man page, whereis <command> locates source for command
* know other commands too
* scp for transferring files
* process is an instance of a computer program currently in execution
* daemon is a background process, such as cron (allows for jobs to be scheduled)
* linux has multiple cores for parallel processing but it really only does one thing at a time
* ps lists the processes, allows you to determine what is currently running
* kill sends signals to kill a program, kill [signal or option] PID(s)
* wget gets program from web servers

10/7/19

* compiled languages are first compiled and then run, runs faster: turns source code into binary
  + examples: C/C++
  + finds more bugs easily because language must be able to turned into a program
* interpreted languages don’t need compilation, interpreter reads program and translates it into internal form: easier to run
  + examples: Python, Javascript
  + additional errors may show up during runtime as it reads in lines one by one
* shell is a program that tkes in commands from the keyboard and gives them to the operating system
  + is cover of the kernel and user’s interface to the OS
* Bourne shell, C shell, bash, etc.
* Shell script is designed to be run on shell, is taken in by an interpreter and all shell commands can be run inside a script
* Shell scripts are useful because they are compatible with command line and are simple
* Unspecified shebang will result in undefined behavior but typically bash by default
* Need space between operands of == for shell script
* If always goes with then
* Use –e flag to detect if file exists, use –f for file
* Put $ in front of variables when assigning it, but in a for loop you aren’t assigning it, so you don’t need a $
* To break, just use break keyword
* Can still use ! for not
* You can pass arguments into a shell script using $1 or $2
* Can have a program run itself if you call $0, causes infinite recursion
* Don’t use “return” in shell scripts because it’s reserved for if the shell script returned correctly or not

10/8/19

* Use find for the homework part
* Use sed command
* \* is a wildcard for any string, ? is for only one character, and [] is used for any elements enclosed (such as [0.9])
* Can test out regex using regexpal.com
* Next lecture will be regex

10/14/19

* Can specify delimiter in print using print(string here, sep=”delimiter”)
* Can also add an ending character that’s added to the last line: print(string here, sep=”delimiter”, end=”last char”)
* To import a module into python, do import module\_name
  + Some modules are available by default, such as os
  + Then to get a function in that module, call it using module\_name.func\_name()
* a = “this is a string”
* print(type(a))

10/15/19

* linux file installation: apt-get (advanced package tool), for Debian and Ubuntu Linux
* old-fashioned process is configure, make, make install
* linux software typically in .tgz or .gz files, use tar –xzvf filename.tar.gz
  + –x for extract, -z for gzip, -v for verbose, -f for file
* two types of compression: lossy, where information is lost, and lossless, where all the info is retained

10/22/19 - Zhaowei

* presentation: 2-3 minutes for introduction, then introduce basic ideas and what makes it different/better
  + give any metrics or experimental results
  + can show video if one exists
* keep amount of words in presentation to a minimum
* no word count guidelines currently for report
* don’t show math derivations, just show intuitive/conceptual steps, don’t show code
* on the final, will be questions about pointers/C and manual memory allocation

10/29/19

* system calls are expensvie because interrupt and trap need to be generated, then need to check if sys call is valid
* different types of system calls: device management, information management, communication
  + device management: need to execute several resources (processes), usually for physics I/O devices
* buffered I/O collects as many bytes as possible into a buffer before making a system call, and this serves to decrease the number of read/write calls, and thus the corresponding overhead

11/5/19

* source code goes thorugh compiler to become object code, then goes through a linker that adjusts any references to fit the memory model of the OS, and then it becomes an executable
* linking can either be dynamic or static
* the linker also takes in code from an object code library to combine with the object file to create an executable
* static libraries are installed into the program before it gets run, while dynamic linked libraries are loaded into the program at startup
* dynamic loaded libraries are loaded and used at any time a program is running (delaying ir further)
* dynamically linked libraries can reduce the amount that gets put together during runtime
* static linking is carried out only once to produce an executable file
* if static libraries get called, linked will copy all the modules
  + extension is usually .a
* dynamic linking allows process to add, remoe, replace, relocate object modules during execution as needed
* during executable creation, only copy some reference information, and then complete linking during loading or running time
* .so extension in unix and .dll in windows
* Shared libraries are good for efficiency and maintaability
* Dynamic linking: executable is smaller, and more flexible for changes
  + Cons: performance (but not too much to loss), errors due to stale/lost references, missing libraries, and wrong library versons
* Dynamic loading: mechanism for loading shared library to memory at runtime
* Still need to load shared library, advantage is to allow startup in the absence of libraries, load libraries lazily (only when needed), and keeps object file small
* Commands for static/dynamic linking
* Big picture: create static/shared library, (ar for static, gcc for dynamic)
* Link that library when compiling the program
  + Gcc, flags can be confusing
* Dynamic loading in your program: dlopen(), dlsym(), dlclose)(
* Automate the process: create a makefile
* Compil libraries with –c (no linking step), and to create dynamic library, use –fPIC flag
  + gcc (-fPIC) –c [file.c] –o [file.o]
  + to create static library, use ar rcs [libxxx.a] file.o
  + to create shared library, use gcc –shared [file.o] [libxxx.so]
* to link library statically/dynamically, use –L [path] –l [xxx] [your\_main.p] –o [exe name]
* [path] is the location of the library, and [xxx] is name of library without prefix lib
  + Thus, if library name is libtest.so, use –ltest
* To make sure linker can find the library, add directory path to $LD\_LIBRARY\_PATH, or use   
  –ldl –Wl,-rpath=[path] when compiling code (no spaces before/after equal sign!)
  + –Wl,rpath adds path to the runtime libraries

11/7/19

* For makefile:

all: main.c engine.o //targets

gcc –O3 –o car //this is the make rule

engine.o: engine.c

gcc –c engine.c //-c is saying don’t compile all the way to executable, just make obj file

* To distribute static libraries, all you need is a header file and an object file
* Header file lists the available functions/interfaces
* Those that write the library are responsible for telling you the function declarations
  + This does not reveal the source code, which is found in the obj file
  + Source code of engine.o is hidden
* Static libraries are no longer common
* Ar can be used to link static libraries together, used for creating archive files
* Ar will give you a drive.a file, where .a suffix indicates an archive
* To create static library: gcc –c –o [executable name] [filenames]
* Ar rcs [archive-name.a]j
* Dynamic libraries: need to make some dynamic links, using ln –sf libhello.so.0 libhello.o
* Then use gcc and –L[link name] to create the program, which will automatically find the missin object/so/a file
* If linking still occurs during runtime, then t’s called dynamic linking
* To tell a particular program to use a certain library, need to add the currnet directory (or directory containing the library) to the LD\_LIBRARY\_PATH vairalbe
* Dynamically loaded library (versus dynamically liked library): same approach, not too much difference and are often interchangeable
* For dynamic loading, use void\* dlopen(…) function
* Also need dlsym (which gives an address to the symbol of the dlopen object), where handle is the value returned from dlopen and symbol is a NULL-terminated string
  + So first do dlopen, (which returns NULL) upon failture, then dlsym, and then dlcose when you’re done (so that you close the module)
  + Use dlerror for error checking, see if any step failed
* To compile, add –ldl flag and specify demo\_dynamic.o
* Then add path to your environment variable and run the program
* Linking is good for time, reduces need to recompile unmofieid source files
* For shared libraries/dynamic linkig, It reduces the binary size and results in a smaller memory footprint because single copy of library code can be shared by all processes
* However, with dynamic linking, loading takes more, the library has to be able to found, and the correct version has to exist
* You can also call a C function from C++:

//insert header guard here

#ifdef \_\_cplusplus

extern “C” {

#endif

//header code goes right here

#ifdef \_\_cplusplus

}

//insert end header guard here

* Name mangling: after C++ is compiled, it will mess up the names of all the functions, and so you need “extern “C”” to make sure this doesn’t occur
* Application for figuring out library linking/usage: library inter-positioning
* You can do compile-time interpositioning: need new functions that are debug versions of the library functions, such as debug\_malloc
  + Debug\_mallocw ill call real malloc, but also print out how many bytes were allocated and starting where
* Once you have it defined, make the object file, and then:
* gcc –DCOMPILE\_TIME –I. –o compile-time main.c debug\_malloc.o
* link time interpositioning: to interchange symbols, trick is using –wrap,[symbol], or –Wl, and it basically tells the linker to resolve wrap as symbol
* see that as \_\_\_real\_symbol is resolved as symbol, and then wrap \_\_wrap\_symbol around it

Declare malloc as \_\_real\_malloc, \_\_real\_free, and then define wrapper functions \_\_wrap\_malloc, \_\_wrap\_free

* load time interpositioning: for when you only have the executable but not the source code, then you can use load time interpositioning to trace calls to functions if you think they exist

11/13/19

* local version control system: organize different versions of a folder on the same machine
* collaboration is difficult because the information is only on your computer
* centralized VCS: have a central server where you store versions of the files, users copy a file over to get it, make changes, and then send it back to the server
* if server has problems, you can lose data, and also, it can be time consuming to download files
* distributed VCS: every machine has a replica of the version history, so you can work even if central server is down
* users can communicate changes directly with each other as well
  + This is the model of Git
* Can use git without github, git focuses on the local aspect of version control
* Git has issues when scaling up to large repositories
* Git basics: checking out code, making changes, staging changes, committing changes, and sharing changes
* Files can exist in 3 states: modified (file changed but not committed), stage (modified and marked to be committed), and committed (stored in database)
* To create a respository: git init, to take code from other repository: git clone
* Branching: git branch <new\_branch\_name>, git checkout <tag/commit> -b < new\_branch\_name>
* Commits: git add <files to stage>, git commit (saves changes to repository)
* Internal structure is a node, where each node contains a version of the file
* Branching allows multiple people to work on a project at once
  + Then, at the end you have to merge the two branches back together, and git also does this automatically
* For merging, if there are conflicts, you’ll need to resolve the conflicts manually
* For information, can use git statuts, diff, log, show, and help
* Git add and git commit are done together often, so can combine it into one command using git commit –a
* Can use git diff to see the differences between files
* Head is a pointer to the latest change you’ve made
  + Head ~n looks back n commits

11/14/19

* To create a directory, use git init
* Git creates snapshots for each version, where it saves the files for each version unless it hasn’t been changed –git keeps all the versions
* To ensure integrity, git calculates a checksum for every single version
* Git has three states: working directory, where you change your files but it’s not recorded by git
* Staging area stores what goes to the next commit, want to tell git that you want these files to be a part of the next version
* Then you commit to save the changes to the repository
* To add file to staging area, use git add <file\_name>, then use git commit to move files from staging area to repository
* Use .gitignore to stop tracking certain files
* Git status to get current status of local branch with respect to HEAD, git diff for which files are different, and git log to see past commits
* Use git init to start a directory - this creates a hidden directory called .git that tracks everything
  + Repository is located in the .git repository
* Git status tracks what’s in a staging area, what’s untracked, and etc.
* To add the file to track it, use git add
  + This moves the file to the staging area, and now it’s waiting to be committed
* When you stage a file, you are only staging the file at that point, and if you change it afterwards, those new changes will not be staged
  + Thus, file can both be in staged area and working directory
  + To add the newest version, use git add again
* Then use git commit –m “commit msg here” to commit, then it’s commited and you get a checksum
* To find a checksum, use git check, and then to find what changes were made for that checksum, use git show
* To roll it back, use git reset –hard <hash\_value>
  + This gets rid of all the older versions
* Can use git reflog to see what operations have been made
  + Can use this to find the hash value from the future, when then allows you to roll it forward
* Use git diff to see in detail what the differences are between files
* To remove local changes, do git checkout – B
  + This makes B go back to the previous commited version
* Cannot undo modifications to local changes due to git!
  + Thus, staging area is good because it allows you to save incremental changes to the staging area, like saving a file (so that it won’t get erased by git)
  + Also, if you don’t want to commit all your changes, you can use the staging area to do this
* Git checkout is only for files that have been committed
* Git reset HEAD B moves B from the staging area back to the working directory (unstage it)
* Git branching: important for allowing multiple people to work on something: multiple branches, and then merge it together
* Git branch shows branches
* Use git merge to put two branches together
* For big projects: merge all changes to development branch, which is then tested before it goes to the master branch
  + Dev branch includes all changes merged together
  + Merge conflicts may occur! Some lines will have to be resolved manually, choose which line is correct
* To create a branch: use git checkout to switch to a new branch: git checkout –b dev
  + This creates a new branch called dev and will switch to that branch
  + Then, to commit, just do the usual git commit –m
  + Everything on the branch is independent of the other branches
* Then, to merge branches together, use git merge <branch\_name>
  + Make sure you switch to the branch you want to merge into first!
* Then they will have the same state
* How to deal with conflicts in code? Occurs when you try to change the same line to different things in two different repositories that you are merging together
* To fix: open the file, and you will see the conflicting lines, then delete everything you don’t want
* Can use other commands to make the branch structure look nicer
* Easier way to look for commits: add tags
* Git tag second <hash\_val>, and this adds a tag to that particular hash
  + Then you can search for that commit by the tag: git show <tag\_name>
* To change patches, use git format-patch -l <hash\_name>
  + This creates a patch using a specific commit
* To work with a repository: use git push to push changes to a remote repository
* To get changes from remote, use git pull
* Use git clone to copy from a remote repository
* Using git branch –vv, you can see which local branches are associated with a remote branch
* Git pull is the same as getting changes from the origin and merging it into the master branch

11/19/19

* Communication over the internet: need confidentiality, data integrity, authentication, and authorization
* SSH: called secure shell, used to remotely access shell, and is the successor to telnet, has an encrypted and better authenticated ssession
* Unencrypted session: the password can be directly seen on the network, but with encrypted, it cannot be viewed during transmittal
* Symmetric encryption: shared key/secret key, key used to encrypt is the same as using to decrypt, which is a problem once you get access to tat key
* Assymetic encryption: public and private keys
* Only creator knows the relation, private key cannot be derived from public key
* Data encrypted by public key can only be decrypted with private key
* Public key can be seen by anyone but private key is never published
* Anyone can use public key to encrypt, and then you use your own private key to decrypt the information
* Early ciphers: shift everything a certain number of letters, but the issue is key distribution, where the secret can be compromised
* Thus, with public key, anyone can encrypt it, but then they cannot decrypt it
* Many algorithms rely on factoring large integers into their prime numbers
* P versus NP: certain problems cannot be easily solved in a short amount of time
  + If P=NP, then decryption will become too easy and thus lose its value
* Ssh: client ssh’s into a remote server; if the first time talking to this server, host validation occurs
  + This is because ssh doesn’t know this host yet, and so it shows the hostname, IP address, and RSA fingerprint so that you’re sure it’s the correct hose
  + After saving, the public key is saved in ~/.ssh.known\_hosts
* However, how are you sure the public key is trustworthy? Use an independent certification authority
* Next time the client connects to the server, if the public key doesn’t match, then it says that it might be someone changing the network
* Man-in-the-middle attacks can occur
* Hsot validation continued: client asks serve to prove that it’s the owner of the public key using assemetric encryption
* Encrypt the message with the public key ad then decrppt with the private key
* Session encryption: client and server agree on a session
* Digital signature: need to generate a message digest that will change if any bit is changed, then create a deigital signature, which is created using the sender’s private key, and then you send the two together and send it to the receiver
* Message digest is a summary of the message that will be transmitted, make sure it produces a different hash
* Receiver: compare the digest to the one generated by the receiver, and if they are not exactly the same, then the message has been tampered with
* Recover the message digest: decrypt the digital signature, and then use the same algorithm to generate a MD of the received image, and compare the two
* Key points is when to use public key, when to use private key, which determines how they determine who gets to decrypt/encrypt why
* However, signature matching is a bit different: why was private key of sender used to encrypt the MD? Why would it be useless to use the sender’s public key?
* Lab 8: using openSSH to securely log in to each other’s computers, generating key pairs, making logins convenient by being able to only type your passphrase once and then use ssh to connect to any other hose without using passwords or passphrases
* Use port forwarding to a run a command on a remote host that displays on your own host
* For lab, make sure you install appropriate packages: openssh-server, and openssh-client
* Run ifconfig to get the current network info, hostname –I to get IP address of machine, ping <ip\_addr> to ping a certain ip address

11/21/19

* SSH, public/private keys are only working if both ends are trusted
  + Thus, if you trust a malicious actor, then it defeats the purpose of this protocol
  + Need a degree of trust when you establish contact with a person
* For final, make sure you are clear about the principles of encryption, etc.
* Symmetric encryption keys are used to maintain info between sessions, as a session key is generated that only the client and server knows, but if it is found out, then this is a failure of the system, and then the messages will be exposed
* Message digest is the other source of the problem: need to make sure MD is not tampered with
* Message digest algorithm must be very hard to reversible so that you can’t undo MD, then encrypt again using private key, then decypt using public key, and then compare the result to the result generated by the MD algorithm (known to both parties), if not equal, then the MD has been tampered with
* Also, trusting the public key can be an issue, as you can mistakenly decrypt an MD the wrong way and end up having it match the encrypted MD created using the message digest algorithm
  + You are relying on an external source to confirm that the public keys are trustworthy, so it’s a potential point of failure
* Detached versus undetached signatures:
* Anything transmitted over the internet needs a signature so that people know that the document is strue
* Like an electronic stamp or seal, almost like written signature but with more guarantees
* Is appended to the document, or sent separetly (detached signature)
* Ensures document was not changed during transmission, and thus prevents tampering and/or impersonation
* Message gets hashed into a MD, and then encrypt with priate key to form cipher, and this resulting item is the digital signature
* Sender’s public key is available on the network
* Receiver has received message, cipher, and the sender’s public key
* First step is to use public key to decrypt cipher back into MD, applies the hash function onto the message to generate a MD
* Then compare the two MD’s: if the same, then nothing bad happened; if not, then tampering might have occurred
* Thus, system breaks down if the actor is malicious: the message and key are both malicious

11/25/19

* Creating a remote repository: so that you can share your work
* Git is a distributed file control system
* To create a repository on github, just click new, add repository name and description
* On the git side, make a new git project: git init
  + This helps manage the local directory
* Then create a file, git add ., git commit –m “First commit”
* Now how to link this git repository to github? Need to tell them that it’s associated with each other
* Use the command git remote add origin <github link>, thus adding the remote repository to the local
* Remote needs a name, by default it is origin
* You can have multiple remote repositories connected to one local git repository
* Git push is a command for pushing changes from local repository to remote repository
  + If you are doing it for the first time, need to add –u <remote\_repo\_name> <branch\_name>
* What if you are a new collaborator, and you want to get the entire history of the remove repository to your local computer?
* Use git clone: click clone with HTTPS, and then in your folder, use:
  + git clone <git\_address>
  + this downloads the entire repository and its history
* git fetch: this synchronizes your orign/master branch with the server
  + this actually creates a new branch that points to the HEAD on the remote
* when you do git clone, it actually creates the new branch and then merges it to the new branch
  + thus, you have your own local branch, and then when you upload it, it tries to make it synchronize to the remote branch
* these are called remote-tracking brnaches: are references to remote brnches
* use git branch –a to see all branches
* -vv to see correlation, how a remote tracking branch is connected to a local branch
* As a result, you can be behind a certain number of commit compared to the remote branch
  + How to solve this issue: use git fetch
  + Git fetch downloads everything from remote branch to the local remote tracking branch
  + But this does not change your local branch!
* The branch that it tracks is called the upstream branch
* To create such a tracking branch, use git checkout –b <branch> <remote>/<branch>
* Create new branch called dev on github, but local doesn’t know about it
* First git fetch, then you can see the remote branch is now a remote tracking branch with git branch –a, but currently, there’s no branch tracking this new branch
* Now, use git checkout –b dev origin/dev
* Workflow: commit on local branch, then sync with remote tracking branch, and then push changes to server so that the remote tracking branch is in sync
* Switch branches back to mater: git checkout master
* Then when you create a new file, use git push to push all changes locally on this single branch to the remote branch that it is linked to
  + Actually, local pushes to remote tracking branch (master to origin/master), and then remote tracking branch pushes to the remote repo (origin/master to origin on github)
* To synchronize to a directory without anything:
* git fetch, which downloads the new branch (origin/dev)
* Then git merge origin/master, which merges from origin/master to your local repository
* To simplify and combine these commands into one, use git pull: this combines git fetch and git merge into one command

11/26/19

* A branch corresponds to a certain state of the software with some sort of features
* Various ways of doing merge: fast-forward merge, 3-way merge, and 3-way merge with conflicts (review these!)
* Whenever you try to merge something, there may be conflicts: when 2 different branches are making changes to the same location
  + Git system cannot deal with this, so to be on the safe side, users must resolve these changes manually
* Fast forward merge is a special case of merge where you merge a branch tha’s ahead of your checked out branch
  + Can only do this when the master had no commits so it never diverged from the branch
  + Then delete the branch (helps simplify/cleanup things)
* Remote tracking branches: these are references to remote branches, and git automatically updates them whenever you communicate with the remote repo through a network
* They’re designed to reflect the last knonw state of the corresponding remote branches
* Usually take the form <remote\_name>/<branch\_name>
* Remote tracking branches will update whenever there is a change
* Local repository can be tracking multiple remotes at once, so that’s why the remote\_name is needed
* Git clone clones a remote repository into a local repository
* Local branch can diverge from the master, but the remote branch tracks the last commit that was synced up to the remote branch
* Git fetch: this allows for the remote tracking branch to get an up to date view of the master branch in the remote server
* Basically, the remote master may be advanced, but the remote tracking branch is behind, so now, when you run fetch, what it does is create a new branch with your local commits, and then advances the local master so that it is up to date with the remote master
* When you have multiple remotes, you can add them using git remote add <name> <url>
* There is a hybrid type of repository called a distributed repository, which is what git has
* Checking out a local branch from a remote tracking brnch will automatically create a tracking branch, and the branch that it tracks is the upstream branch
  + Tracking branch basically tells local branch that it has a replica in a remote repo
* Commands: git checkout –b <branch\_name> <remote\_name>/<branch\_name>
* Git checkout –b b1 origin/b1 will create a local branch b1 tht’s a copy of and tracks origin/b1
* Git fetch only fetches the changes but doesn’t change the local workig repository
* Git pull is a command that combines git getch and git merge, so that remote changes are fetched and those changes are also incorporated itno the local working directory
* By default, git pull pulls from the branch that the current local branch is tracking
* Git push serves to update the remote branch with the local branch
* Git push is a shortcut for git push origin b1:b1, which means pushig local b1 branch onto borigin’s (remote) b1 branch
* Git rebase: this serves to cut away changes from one branch and bring them over into another branch
* Rebase versus merge: you should rebase branches that have not yet been published, as if you rebase a published branch, then it will mess up history and thus other’s people work that depends on the current state of the branch
* Otherwise, use merge
* Showing revisions: can use the hash to find a specific commit
* ^ symbol at end of reference refers to the parent of that commit:
* For merge commits, the ^ refers to its first parent, and ^2 refers to its second parent
* HEAD points to the most recent commit, so HEAD^ ad the most recent commit’s ^ is the same
* Revisions election: ancestry references
* A tilde ~ also refer to the fist parent, so ^ and ~ are the same
* Commit ranges: hash1..has2 selects range (hash1, hash2)
* Rewriting history/commits: can reset them and take back to staging area: git commit –amend
  + Can use this to change commit content and commit message
* Should only do this when the changes will not affect the commit that has already been pushed to remote
* Also, can use git rebase –I HEAD-3, to rebase the last 3 commits into the 4th oldest commit (merges them together)
* Commits are listed in reverse order compared to that of git log!
  + Then a text interface will open and you can select what to keep/remove
* Thus, there are three trees overall: the HEAD, which points to the last commit snapshot
* Index, which points to the next commit snapshot, aka staging area
* Finally, working directory, which has the files that you have
* Git reset: --soft option only moves the HEAD to an earlier commit, and then stops at this point
* Thus, this shows any changes after that as being staged
* --mixed option: also changing the index, which also means unstaging everything, but the working directory is not changed
* --hard option removes commit from history and staging area, and also changes the local working directory, thus causing all changes to be lost!
* At the file level, git reset does not move the head pointer

11/27/19

* Create tree in git: git write-tree
* Before doing this, index every file that you want to add to this tree
* Creates a tree object
* Use git cat-file –p <hash>
* Index the file: git update-index –add –cacheinfo 100644 <hash> <file\_name>
* This tree is within the .git/objects file
* To show it, use git cat-file –p
* To check of a object, use –t with git cat-file
* An object can be a blob or a tree
* If file is already in the directory, can just go git update-index --add new.txt
* Can multiple files into a staging area to make a new tree: add the old one too
* Then, using git write-tree, you can have a new tree containing those objects
* Creating a tree node with another tree as its child
* Use git read-tree –prefix=bak <tree\_node\_hash>: read-tree appends another tree onto it
* Then git write-tree, which creates a new tree
* Then, using git cat-file –p <tree\_hash> lists all the nodes
* Then, use the listed hashes to explore any subtrees
* To create commit, specify a tree and your commit message
  + The commit basically points to a tree object
* To commit a tree, use git commit-tree <tree\_hash>
  + This creates anoter oject, called commit object
  + Again, use git cat-file –p <commit\_hash> to see details about the commit, including the hash of the tree object
* To make another treel can simply start over process
* Don’t delete anything in your .git repository because you will lose all progress
* No new filse are created, but new
* Maser stores the Ssh: then git log lists the branches
* Then from this, can naviate’back to children
* Ls lists the branhches tahathe prahcnes you made compaer to oline iA
* The tag is a file that points to a specific commit
* Lab 9 notes: answer a few questions about eggert’s emacs repo
* 1. Check disk usage
* 2. Use git branch, the count
* Use git log, git commit, etc.
* Homework is to essentially create a git map
  + Go up the directories until you find a .git directory
* Need to able to decrypt the git objects manually using python
  + Read the section on “object storage” on the git online docs
* When choosing parents for commit, choose the parent with the longest chain of commits going up back to the root
* Topological ordering for this lab: print out newest at the top, and then print out the oldest last