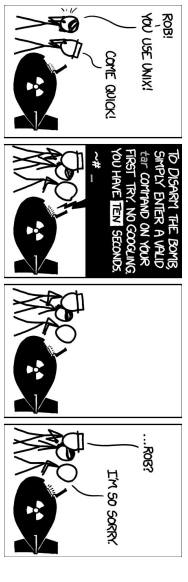
HELP fill in TODOs or weeks where your TA graded (break out sample code to separate pages / print out your own assignments/labs) [proofread notes, ensure correctness (will be double checked)]

Week 1: Lab 6: Grader: **Nandan Atul Parikh: System Basics**

- Operating System: manages memory, processes, other softwares & hardwares (layer between application & hardware)
- Debian GNU/Linux
    - Kernel
    - Shell
    - Programs
- Kernel
    - Core of OS
    - Allocates CPU time & memory to programs
    - Interfaces software to hardware
    - Allows inter-process communication
- Interactive Shell
    - Interface between user & kernel (also see week 2)
    - **<up arrow>**: previous command
    - **<tab>**: auto-complete
    - **!!**: run previous command
    - **![*str*]**: run previous command that contains *str*
    - Moving around
        - **pwd:** print working directory
        - **cd:** change directory
            - ~ home
            - . current
            - / root
            - .. parent
        - **mv:** move file from DIR1 to DIR2
            - -f: no prompt before overwrite
            - -i: interactive
            - -n: do not overwrite
        - **cp:** copy a file
            - -l: hard link files
            - -L: follow symbolic links in source folder
        - **rm:** remove a file
            - Delete files permanently
            - -r: for directories, remove recursively
        - **mkdir/rmdir: make directory / remove directory**
        - **ls:** list contents
            - -d: list only directories
            - -s: show the size of each file, in blocks
            - -a: list all the files including the hidden files
            - -A: list all files EXCEPT . and ..
            - -l: list the details of each file including permission info
                - drwxrw-r-- format
                - 1 bit: file type (- regular, d directory, l symlink)
                - 2~4 bits: user permission
                - 5~7 bits: group permission
                - 8~10 bits: others (everyone) permission
                - Number of hard links, User(owner) name, Group name, Size, Date/time last modified, Filename
            - -t: list the files based on files' modification time
            - -r: list the files in the reversed sorting order
            - -i: list inodes (matching inodes means hard link)
    - File permissions
        - **ln, readlink:** create link, read link
            - ln [*source*] [*hardlink*]: creates hard link
            - ln -s [*src*] [*softlink*]: create "pointer" to file
            - readlink
        - **touch:** update access & modification time

        - If file does not exist, creates file
    - **chmod:** change modifications
        - u: user g: group o: others a: all +: add perm -: minus perm
        - r: read perm w: write perm: x: execution perm t: sticky bit (files in directory can only be modified by file creator / owner of dir) s: setuid & run executable with permission of owner / group (setgid)
        - Numeric mod: NNNN: left to right 1st: sticky bit(1), setgid(2), setuid(4); 2nd: user perms (execute(1), write(2), write and execute(3), read(4), read and execute(5), read and write(6), full(7)); 3ed: user's group; 4th: everyone else (3ed 4th use same notation as second)
- Shell commands
    - **man: main linux documents**
        - man [n] [keyword]: show nth page of keyword page
            - 1: user command
            - 2: system calls
            - 3: c functions
            - 7: other
        - -k [*keyword*]: search for keyword
    - **cat [file]: show file contents**
    - **od [file]: dump file in octal & other formats**
    - **head/tail [file]: print first/last 10 lines**
    - **du [folder]: estimation of file space (default unit 1024 bytes)**
        - will recursively look through all folders
        - -s: summary -h: human readable format
    - **ps: show processes**
        - -a: show all process of user
        - -au: show all processes of all users
        - -aux: all processes, even non-user users
    - **kill [process]:**
        - -s SIGNAL: send SIGNAL to kill PROCESS
    - **diff:**
        - -u: unified context, used for patches
    - **comm FILE1 FILE2 (compare line by line)**
        - -1, -2, -3: suppress column: (FILE1 unique, FILE2 unique, both have)
        - Each line of output may have: -1 zero tabs, -2 one tab, -3 two tabs
    - **wc [option] [file]:** show line count, word count, & character count
        - If there is no file name, processes data from stdin
        - -l: only showing line count
        - -w: only showing word count
        - -c: only showing character count
    - **sort:** outputs input (either STDIN or file), sorted
        - --parallel (week 7: multiprocessing)
        - -u: unique sort
    - **echo:** print STDIN (NOT FILE)
    - **find [dir] [optional string]: find string in directory**
        - type [dfl..]: find only of type: (d)irectory, regular (f)ile, sym(l)ink
        - -executable: searches for executables
        - -L: follow symlinks; -H: inverse
        - -mtime [n]: accessed within the last n days
        - -perm [numerical]: files that match numerical permissions
        - -name: name of file
        - -user: owner of file
        - -maxdepth: how many levels below to go into
    - **sort:**
        - -g: based on pure numerical value
        - -u: sort uniquely
    - **comm [OPTION]...FILE1 FILE2**: compare two sorted files line by line
    - **tr [SET1] [SET2]: stream translate**
        - Ex: echo "12345" | tr "12" "ab" => ab345
        - echo “abc” | tr “abc” “12” => 122
        - -s: squeeze repeats of last set into single one
        - -c: complement of SET1



                - -d: delete characters in set 1
                - [/n*]: in set2, expand /n until length of set1
            - **whatis, whereis, which <command>:**
                - whatis: brief summaries of command that follows
                - whereis: src, docs, & binaries of command
                - which: full path of command that would execute
            - **wget <url>: gets a url, copies file into curdir**
            - **locale**
                - show locale information, which governs how certain things are displayed
                - set using `export LC_ALL='C'` (ASCII) or `export LC_ALL='en.US_UTF8'`
            - **tar**
                - -x: extract files from archive
                - -r: append file to archive
                - -c: create new archive
                - -z: filter through gzip (compress/decompress arxiv)
                - -j: filter through bzip (different compression)
                - -f: specify file
                - -C: specify folder to extract to
    - Special tokens
        - Redirection operators (evaled left to right in command)
            - <: feed input from file to command
            - >: feed output from command to file (<> do both)
                - 1>: default - redirect stdout to file
                - 2>: redirect stderr to file
                - 2>&1: squash stderr into stdout (do after redirect)
            - >>: append output to file
        - Control operators:
            - [command]&: run command in background (switch back using fg)
            - [] && []: logical AND
            - [command] | [command2]: output of command is input of command2
            - |&: 2>&1 | (both stderr & stdout are passed to command2)
            - [] || []: logical OR
- Files/Processes
    - Process: executing program by PID
    - File: collection of data
        - Includes symlinks, executables, directory, devices
    - Linux filesystem layout: tree based
- EMACS commands:
    - C-h t (help tutorial)
    - C-h a <string> (show commands matching string)
    - C-h k <key> (describe the function a key runs)
    - C-h f (describe a function)
    - C-h m (get mode specific information)
    - C-x C-s (saving a file)
    - C-x C-f <filename> (opening existing/creating new file)
    - C-p, C-n, C-b, C-f (move up, down, left, right)
    - C-a, C-e (move to beginning, end of line)
    - M-<, M-> (move to first line of text, move to last line of text)
    - M-g g [number] (move to particular line number)
    - C-Space (set marker)
    - C-w (cut/kill)
    - M-w (copy)
    - C-y (paste/yank)
    - C-x u (undo command/edit)
    - C-s (search, repeat to cycle)
    - C-r (cycle backwards)
    - M-% [source] enter [dest] (replace)
    - C-k (erase from current cursor to end of line)
    - C-x C-b (list all the buffers)
    - C-x C-w [filename] (save current file w/ filename)
    - C-x h (save entire buffer)

Week 2: Lab 4: Grader: Jeremy Rotman: Shell scripting

- Shell Programming
  - Common shells: bash, sh, csh, ksh
  - Shell script - first line: #! bin/bash (#! bin/sh)
  - test *expr* or [*expr*] or [[*expr*]] (latter is preferred)
    - logical test (e.g. use in if)
    - file tests
      - -e (name): file with name exists
      - -f (name): file w/ name is regular
      - -h (name): file w/ name is symlink
      - -d (name): file w/ name is directory
      - -s (name): file exists and is not empty
    - string test
      - -z: string length is zero (inverse -n)
      - = : strings are exactly equal (RHS in quotes)
    - numeric tests: ARG1 [opt] ARG2
      - -eq: equals -ne: not equal
      - -lt: less than -le less or equal (equiv. -gt, -ge)
  - Locales
    - Set of Parameters that define a user's cultural preferences: language, country, and other area specific things
    - LC_COLLATE determines the order for comparing and sorting
    - =C: compare ascii values (accented characters come later)
    - =enUTF8: similar (eng alphabet comes first)
    - =other UTF8 (eg frUTF8): in order of french alphabet (Hon Hon Hon)
  - wildcards (globbing patterns)
    - ?: matches single character
    - *: matches zero+ characters after
    - []: matches any one of the characters between the brackets
    - \: escape whatever came before (a la ''); \\ escapes itself (whoa meta)
    - .:
      - file specifier (curdir is '.') ('./script' executes script, but 'cat ./script' outputs content of script ⇐ used to escape files that start with '-')
      - alias to execute ('. script') also executes script
    - see regex for more oof
  - Variables & arithmetic
    - **NO SPACES BETWEEN ASSIGNMENT EQUALS SIGN**
    - var='this'
    - echo $var
    - $ variables:
      - $1, $2....positional arguments ($0 is name of script)
      - $@: array of positional arguments (wrap in "")
      - $*: all positional arguments separated by IFS
      - $?: exit status of the most recent command
      - $IFS: separator (space), used for slicing
      - $HOME: home directory
      - $PATH: all directories to look for executables in, from left to right
    - arithmetic
      - `expr *something*`: interpret math
      - ((*something*)): also interpret math
      - so e.g. a=$(($var + 3))
  - execution tracing
    - set -x: prints out every line while it is being executed
    - set +x: to turn it off
  - trap
    - usage: trap (command) (signal (numerical or SIG***)
    - runs command at signal
    - C system calls basically set trap (system call) (custom signal) (see w5)
  - quotes
    - "": escape ${}, ``

        - ``: expand as shell commands (run commands)
        - '': literal meaning (expand nothing)
    - program control TODO
        - if, for, while else
        - While:
            - While loop

```
#!/bin/sh
COUNT=6
while [ $COUNT -gt 0 ]
do

echo “Value of count is: $COUNT”
(( COUNT=COUNT-1 ))

done
```

            - For loop

```
#!/bin/sh
temp=`ls`
for f in $temp
do

echo $f

done
```

- Regular expressions: https://alf.nu/RegexGolf, https://xkcd.com/1313/
    - Regex is a special text string for describing a certain search pattern
        - ^ is beginning of a line (ex: ^tolstoy: the 7 letters tolstoy at the beginning of a line)
        - $ is end of a line (ex: tolstoy$: the 7 letters tolstoy at the end of a line)
        - / means turn off meaning of next character (escape)
        - [] is match any of enclosed characters; use - for range (ex: [Tt]olstoy: either the 7 letters tolstoy or 7 letters Tolstoy anywhere on a line)
        - . matches any single character (ex: tol.toy)
        - * matches 0 or more characters of previous character/expression
        - + matches one or more characters of previous character/expression
- syntax (capture groups)
    - TODO
- sed, grep, BRE, ERE, PCRE
    - grep: A Unix command to search files/text for the occurrence of a string of characters that matches a specified pattern
        - Usage: grep [option(s)] pattern [file(s)]
        - Options:
            - -c: suppress normal output; print count of matching lines
            - -n: precedes output with line number
            - -v: invert match
        - Examples:
            - grep –r ‘*.txt’ *
            - grep –c ‘line’ file.txt
            - grep –n ‘line’ file.txt
    - sed: a stream editor used to perform basic text transformations on an input stream
        - Printing:
            - sed –n ‘1p’ sedFile.txt (prints line 1)
            - sed –n ‘1,5p’ sedFile.txt (prints line 1 through 5)
            - sed –n ‘1~2p’ sedFile.txt (prints every 2nd line starting with the 1st)
            - -n suppresses output (not sure what that means?)
        - Deleting:
            - sed '1~2d' sedFile.txt (deletes every 2nd line starting with the 1st)
        - Substituting text: s/regex/replacement/flags
            - sed 's/cat/dog/' file.txt
            - sed 's/cat/dog/g' file.txt (g is a flag)

            - -i edits in place instead of printing to standard output
    - Awk: is more than a command; it's a programming language by itself; Utility/language for data extraction
        - Usage: awk '/search pattern/ {Actions}' file
        - awk '{print;}' file.txt // print the file line by line; default behaviour
        - awk '/Hello/ {print;}' file.txt // prints lines which matches Hello
        - awk '{print $1,$2;}' file.txt // prints only specific fields
        - Awk 'BEGIN{x="cs35l is torture"; print toupper(x)}' ⇒ CS35L IS TORTURE
            - If BEGIN is not stated then awk will ask for user input and ignore pre-defined x value

Week 3: Lab 1: Grader: Kedar Narayan Deshpande: Modifying/rewriting software

- Diff/patch stuff is in week 9, c compilation stuff is in week 4
- When you receive a tarball file (.tgz) (.gz)
    - tar -xzvf filename.tar.gz
        - Option -x: --extract
        - Option -z: --gzip
        - Option -v: --verbose
        - Option -f: --file
- Building from source
    - ./configure:
        - every script is different (basically the project writes a shell script)
        - --prefix: specifies where make install puts files (usually /usr/)
        - checks capabilities of the machine, generates a makefile
    - make:
        - requires 'Makefile' to run
        - compiles the source (see week 7) and creates executables in current
        - temporary directory
        - has different targets specified by make [TARGET]
        - make ⇒ first target, usually make all: makes everything specified (may call other makefiles)
        - make clean: removes all binaries
        - make install: moves binaries to prefix/bin
        - allows the use of macros (like C macros) that can be changed quickly (week 7)
- Python
    - scripting language
    - object-oriented
        - classes
        - functions
    - compiled & interpreted
        - Compiled:
            - code is reduced to machine-specific instructions before being saved as an executable (at compile time)
            - faster than interpreted code.
        - Interpreted:
            - code is read "as is". Reduced to machine-language at runtime.
            - more versatile than compiled code.
        - Python is both: reduced to Bytecode first, then interpreted by the Python Interpreter
    - very popular for its "ease of use"
        - constantly uses English keywords instead of punctuation symbols like other languages
        - fewer syntactical rules
        - automatic garbage collection
        - easy integration with other languages
    - has two modes:
        - interactive: run single commands on the python shell
        - script mode: write a set of a commands to execute. This is where you write your programs.
- Coding in Python
    - types of variables do not need to be specified; case sensitive
        - x = 10 # interpreted as an integer

        - name = “Paul Eggert”   # interpreted as a string
        - my_gpa = 2.01   # interpreted as a floating point
    - Python list:
        - dynamic: can change in size
        - heterogenous: can contain different types
        - access items just like arrays in c: list[0] => first item in list
    - Python dictionary:
        - similar to a hash table
        - provide a key/value pair
        - instantiated using curly braces: dict = {...}
        - keys must be unique, values not necessarily
    - for loops:
        - unlike c, for loops iterate through items in a list
        - to create a list of numbers, use range(numOfItems). List starts at 0 and goes to numOfItems-1.
    - argparse vs optparse
        - both are libraries to take input from command-line
        - argparse is better than optparse for the most part. We were advised to use argparse in the assignment

Week 4: Lab 2: Grader: Vignesh Sairaj: C programming/debugging

- TODO: c programming if you really want to, struct definitions & memory alignment
- c compilation/linking process
    - gcc
        - -Wall: lots & lots of warning & errors
        - -Werror: make all warnings errors
        - -g: includes debug symbols (needed for gdb to know what it’s doing)
        - -o: specify name of output file
        - -c: compile without linking (stop after assembler)
        - -shared: tells linker to build a -fPIC version of object code for dynamic linker
        - -On: turn on optimization (0, 1, 2, 3, s, g)
            - 0,1,2,3: no, lowest, medium, highest optimizations
            - g: produce more debugging info than -O0
            - s: optimize for size
        - -fPIC: make position independent code (used when making shared libraries)
            - quick explanation: if non PIC, compiler memory moves go directly to memory address; this wouldn't work with different programs, because the memory space may be taken up in another program. this is why we use -fPIC with -c
        - -l[library]: tell linker to search for [library] when linking
    - Process: preprocessor => compiler => assembler => object code => linker => executable
        - preprocessor: substitutes macros (e.g. #include, #define)
        - compiler: translates C code to assembly code (that's missing components)
        - assembler: makes machine code (this is almost runnable, object .o files)
        - linker: takes machine code & "links" other libraries needed, outputs binary
- stack & heap
    - stack:
        - organized into frames that are pushed/popped from computer memory based on calls to function
        - local variables, parameters, return of function are stored in the frame of the stack (any declaration of local variable goes into the stack)
        - too many function calls => too many stack pushed => stack overflow
        - grows to lower memory address from top memory addresses
    - registers:
        - contexts to perform arithmetic (take data from stack/heap)
    - heap:
        - malloc() uses this space; much larger, but slower (dynamically allocated)
        - must be freed afterwards! or else memory may be lost
        - grows up from lower memory addresses
- malloc(), realloc(), free()
    - void* can be cast to any pointer; all malloc & realloc return NULL if failed
        - good way to check (avoid deref null ptr): if (ptr && ptr[5]){

        - void* malloc(size_t size): allocates size bytes of memory, returns pointer to beginning
            - store 40 integers (integers being 2 bytes) => int* ptr = (int*)malloc(80)
        - void* realloc(void* ptr, size_t size): reallocate to size bytes of memory, given address start ptr (cast to void* ptr); size is absolute size
            - realloc might move the ptr (possible original ptr location doesn't have enough space to grow); important to capture return ptr, data will be the same but not necessarily in the same location
        - void free(void* ptr)
            - frees up the memory block originally allocated by malloc/realloc
    - debuggers/tools: gdb, strace, valgrind (usage)
        - strace [./executable]
            - -o: output to file (good when trace program need both in & out redirection)
            - -c: count calls & time & report summary on exit
        - valgrind --leak-check=full [./executable]
            - verifies that heap allocs are kept in variables, & are correctly dealloced afterwards
    - gets() vs fgets(): gets() was deprecated and then removed from C programming specification as of C11 due to returning values with no regard to buffer overflow. fgets() solved this, but programmer must take into account endline at end of stream.
    - qsort(void* base, size_t num, size_t size, int (*compar)(const void*, const void*))
        - Base - pointer to start of array to be sorted
        - Num - number of items
        - Size - size of each element in each array
        - Compar - ptr to sorting function
            - Since compar takes void*, have to create a use a wrapper function
                - Converts void* to sortable types, then calls your actual sort function
    - Gd

Week 5: Lab 8: Grader: Tanmay Chinchore: System calls debugging

- Userspace vs Kernelspace
    - execution / protection
        - hardware mode-bit: 0 ⇒ kernel, mode 1 ⇒ user mode
        - ensure all processes have fair use of CPU/memory
        - prevent illegal I/O operations (writing to graphics buffer e.g.)
        - prevent illegal overwriting of memory space
    - user mode
        - cpu can only call unprivileged instructions in specific memory
        - exception / error only crashes process
    - kernel mode (trusted memory code)
        - 'sudo cpu': unrestricted, use all instructions, access all memory
        - exception / error fucks up entire system
    - system calls
        - calls that exposes kernel mode functionality to user mode applications
        - are part of the OS kernel -- kernel verifies user can do, then does
        - slower, because proc is interrupted (see trap, week 2), OS kernel takes control (switches context) of cpu & needs to verify that it can do it, performs the action, restores old context, hands control back
        - the system call api (functions in C, C++) invoke the actual system calls
    - types of system calls
        - driver/device management: stdin, stdout, file storage
        - os-manageable information:
            - time, date, etc.
            - this also includes timers, process id numbers, etc.
        - interprocess communication (see week 6), process control
            - creating processes, creating pipes, etc.
    - File descriptors, buffered/unbuffered IO (functions we used for assignment)
        - System calls: (requires <unistd.h>)
            - int **open**(const char* path, int flags)
                - path: string (either relative or absolute path)
                - flags: O_RDONLY, O_WRONLY, O_RDWR, O_CREAT

                - returns an integer *file descriptor* (usual 3 or above, 0,1,2 are stdin, stdout, stderr): returns -1 if error
            - ssize_t **read**(int fildes, void *buf, size_t nbyte)
                - fildes: file descriptor (literally an integer) (how to read stdin)
                - buf: memory address to move from nbytes from fildes to
                - nbyte: bytes to read (remember unsigned char = 1 byte)
                - returns how many bytes it actually read; if -1, then some error happened (notice not reading anything is not an error)
                - this moves the file pointer (i.e. the next time you read, it starts reading from the last place it read before: pread(int, void, size_t, off_t) takes extra off_t integer-like offset value
            - ssize_t **write**(int fildes, void *buf, size_t nbyte)
                - reverse of read: takes nbytes from buf & puts them into fildes, returns how many bytes actually got written
                - similar pwrite with similar off_t type argument which does not move write pointer
            - int **close**(int fildes)
                - closes a file descriptor
                - not sure what happens when you try to close 0, 1, 2: not a good idea, & definitely not needed
        - Library calls: (requires <stdio.h>)
            - TODO: fopen, fclose, etc
            - fopen(cont char *filename, const char *mode): Opens designated filestream to read in as input
                - Modes - “r” == reading, “w” == writing, “a” == appending
                - “r+” - reading and writing (file must exist)
                - “w+” - creates empty file for reading and writing
                - “a+” - opens file for reading and appending
            - fclose(File *stream): Closes designated filestream
            - int getchar() -- gets a character from stdin
            - int putchar(int char) -- writes char to stdout
            - int getc(File *stream) -- gets character from specified stream
            - int putc(int char, File *stream) -- writes char to specified stream
        - Why were stdin/stdout read/write library calls faster than system calls?
            - TODO: explanation using buffers, as well as context switch overhead
    - time (bash keyword, not linux command)
        - real: "wall clock time" literally how much real time passed
        - user: time spent in userspace
        - system: time spent in kernel mode

Week 6: Lab 3: Grader: **Guangyu Zhou: Multithreading**

- Multithreading vs multiprocessing vs multitasking vs hyperthreading
    - multithreading: software split, smallest sequence of instructions
        - hardware threads (i.e. cores): single core can concurrently run separate threads that are doing completely different things
            - Thread(s) per core: 2 (NOTE: These values depend on individual
            - Core(s) per socket: 12 machines)
            - Socket(s): 4
            - Your max thread count is, 4 CPU x 12 cores x 2 threads per core, so 12 x 4 x 2 is 96. Therefore the max thread count is 96 and max core count is 48
        - threads have their own stack, but all share the same heap
    - multitasking: CPU switches quickly (time-slices) between multiple *processes*, highly responsive programs, processes are insulated from each other
    - multiprocessing: two tasks running concurrently, either on multiple CPUs, or a CPU with multiple cores
    - hyperthreading: intel marketing term, splits a physical CPU core into two logical cores (taking advantage of parallelism (see CS33)
- pthread: linux library that abstracts threads
    - type:
        - pthread_t: representation of a pthread
        - pthread_attr_t: thread attributes (not important)

            - void* (*func) (void*) ⇒ this means pointer to func that takes in void* ptr, & returns a void* ptr
        - Functions: must include <pthread.h> & compile with -lpthread
            - int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*func) (void*), void* arg)
                - thread: pointer to pthread_t type, _create puts the newly created thread in that memory space
                - attr: usually pass in NULL; we just need default thread attributes
                - func: *just the name of a previously-defined function that takes in void* ptr, returns void* ptr* (see sample code)
                - arg: single argument to pass to func (this keeps it general)
                - returns 0 on success
            - void pthread_exit(void* retval): TODO: verify correctness
                - returns from the thread (and possibly the program, if this is the last thread)
                - retval: pointer to the return value in memory
                - does not return (thread is done)
                - note: this was not needed; returning from the thread function is just as good (manually put data back into pointer); this is for returning early
            - int pthread_join(pthread_t thread, void** retval)
                - thread: wait for this thread to terminate
                - retval: a *pointer* to the location of the *pointer* of the return value when calling pthread_exit()
                - returns 0 on success
                - note: for the assignment, setting retval as NULL was find (because pthread_exit was never called, & the thread just naturally returned, so there was no return data)
                - good for waiting for all spawned subthreads to return
            - pthread_t **pthread_self**(void)
                - Returns the ID of the calling thread
            - pthread
    - Race conditions, mutex, deadlocking
        - if two threads write to same memory space at the same time, behavior is undefined, but often results in problems (e.g. first thread checks for value, getting ready to write to it, second thread writes to it, then first thread writes to it again (second thread's value lost)
        - solution for 35L: mutex: the conch from *Lord of the Flies*
            - single variable that can only be "held" by a thread at a time (known as "locking")
            - processes working on same variable should lock associated mutex, change variable, then unlock mutex
            - if process cannot lock mutex (because another thread locked it), wait until mutex is unlocked, then continue
        - deadlocking
            - occurs when t1 is waiting for t2 to unlock mutex, t2 is waiting for t1 to unlock mutex (execution has effectively stopped)
            - easy solution: ensure that threads always lock & unlock in the same order
        - pthread mutex syntax:
            - initilization: static: pthread_mutex_t mutvar = PTHREAD_MUTEX_INITIALIZER
            - locking: int pthread_mutex_lock(pthread_mutex_t* mutex)
                - with normal mutex type (inherent to static initializer), blocks thread execution until lock acquired: returns 0 upon successful locking
            - unlocking: int pthread_mutex_unlock(pthread_mutex_t* mutex)
                - tries to unlock mutex, may return EPERM: thread does not own mutex
    - SIMD vs MIMD: single instruction, multiple data or multiple instruction, multiple data
        - simd: dragon boat
        - mimd: soccer
        - TODO

Week 7: Lab 5: Grader: Diyu Zhou: Dynamic Linking

- Source Code -> Compiler -> Object Code -> Linkeder -> Exec
    - Compiler translates programming language into cpu instructions

    - Linker combines object files into single executable
- Static Linking
    - Static Libraries extension: ***.a***
        - Modules referenced copied directly into executable
- Dynamic Libraries Extension: ***.so*** (linux/unix)/ *.**dll** (windows)
    - assignment used elf shared libraries (ld-linux.so shared object is first loaded; *the dynamical loader is dynamically loaded* (by execve)
    - Linking done during loading/running time
- Dynamic Loading Functions: (require <dlfcn.h>, link with -ldl)
    - void* dlopen(const char* filename, int flag)
        - makes shared object file accessible to program
        - flag is typically RTLD_NOW (bind all undefined symbols)
        - returns void* handle (literally, declare void* handle and assign dlopen)
    - int dlclose(void* handle): closes handle shared object
        - return zero on succ
    - void* dlsym(void* handle, const char* symbol)
        - try to find symbol within handle, a dlopen() returned library
        - return the address to whatever was found
            - cast pointer to pointer of correct data / function type, proceed
            - returns null if nothing found
    - char* dlerror(void)
        - return in a human readable format, the most recent error caused by one of the functions above
        - returns NULL if nothing happened
- Advantages of Dynamic Linking
    - Smaller executable file
    - When library is changed, no need to recompile code that references it
    - Multiple programs can access same .so
- Disadvantages of Dynamic Linking
    - Need to load shared objects
    - Need to resolve addresses
        - Dynamic Library might be missing
        - Wrong version of library
- Makefile
    - Construction idea: "write top to bottom, compile bottom to top"
        - start with end target, specify objects needed to link it together
        - for each object, specify how to assemble/compile those objects
        - use macros: DEFINE="-flag -flag2", command $(DEFINE) ..
        - clean: remove all object files/binaries
        - .PHONY: specify that next target is not a file target
            - .PHONY clean \n clean: would specify the target clean, not file clean

Week 8: Eggert: SSH & Security

- ssh
    - -X: Allows for X11 forwarding, ex: you can now use GUI windows
- gpg key signing
    - Web of trust (tbh just read this, and this, and this?)
    - what it actually does
        - TODO

Week 9: Lab 7: Grader: **Shiv Dalmia: change management**

- git: the ~~stupid~~ really amazing content tracker
    - local management
        - add
            - Adds files to be tracked in current repo
        - commit
            - Saves version changes in repository by unique SHA ID
        - HEAD
            - commits

        - branches
            - merging/rebase
            - Allows you to work on code separately from live repo
            - Merge changes later into dest branch
        - tags
        - cherry-pick
        - reset
    - remote management
        - push
            - Publishes changes on branch to remote
        - pull
            - Get most recent version of repository from remote
        - Fetch
    - merge vs rebase
        - merge ties together existing branches of the project history at point of merge, is nondestructive → convoluted project history
        - rebase rewrites the project history → clean project history, but erases info
- patch
    - path file format

Cheat Sheet of git commands: https://education.github.com/git-cheat-sheet-education.pdf

---

Also check out:
Random Knowledge likely to expand exponentially the day before final:
https://paper.dropbox.com/doc/CS35L--AZaPHHpV7v53QKarOONVQO1MAQ-1SjpYigfgb2rMKKDA1OO0

Emacs Cheat Sheet (see lab 4 reference card)
https://www.gnu.org/software/emacs/refcards/pdf/refcard.pdf

GDB CHEAT SHEET

GDB Refcard: http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

- Starting GDB
  - gdb
    - Start without file
  - gdb <executable>
    - Debug specific executable
- Location formats (referred to now on as <location>)
  - Places to give to the commands below
    - linenumber
    - filename:linenumber
    - function_name
    - file:function_name (duplicate function names)
- Common commands (abbreviations of the command given next to the name)
  - <Press Enter>
    - Repeats the last command – helpful for repeated use of step and next
  - help
    - Gives gdb documentation
    - Usage
      - help <command>
      - help <topic>
  - file <executable>
    - Open specific file to debug (if gdb was opened with no arguments)
  - list <location>, l <location>
    - Without any arguments
      - Prints the first ten source lines
    - Else, print <listsize> lines @ the location
    - Consecutive calls print the next ten source lines
    - set listsize <count>
      - Number of lines printed on list
  - frame, f
    - Print the current stack frame you're in
    - Useful for seeing the current program you're in
  - run, r
    - Run the program from the beginning
    - Usage
      - run <arg1> <arg2> .. <argN>
        - As if giving inputs from stdin
      - Redirects
      - run < input_file
        - As if ./executable < input
      - run > output_file
        - As if ./executable > output
  - continue, c
    - Resume the program after a breakpoint
  - step, s
    - Execute the current source line, “step” into a function
    - Example
      - int result=sum(a+b);
      - printf(“%d\n”,result);
      - Stepping into this line will stop at the first line inside sum
        - int sum(int a,int b){
          - return a+b;
        - }
  - next, n
    - Execute the current source line, but don’t stop until the line has been completed
      - Example
        - int result=sum(a+b);
        - printf(“%d\n”,result);

                - Calling next on the first line will execute the current line, end at the next line
    - until <location>
        - Run until the location
    - backtrace, bt
        - Backtrace of all the stack frames
        - Useful for debugging segmentation faults
            - run
                - Program will stop @ segfault
            - bt
                - See what function/line caused the asdf
    - info <argument>
        - info args
            - The arguments passed to the function you're currently in
        - info locals
            - Information on all the local variables in the function you're currently in
        - info breakpoints (info b)
            - See Breakpoints section
- Breakpoints
    - break <location>, b <location>
        - Set a breakpoint at location
    - info breakpoints, info b
        - Useful information on all your breakpoints
        - Output
            - (gdb) info b
            - Num Type Disp Enb Address What
            - 1 breakpoint keep y 0x000000000040052c in main at test.c:1
    - delete
        - Deletes **all** breakpoints
    - delete <breakpoint_number>
        - Delete specific breakpoint (get number from info breakpoints)
    - clear <location>
        - Clear the breakpoint @ location

Makefile sample

```
all : shop # usually first -- this is the full program
shop : lookup_ item.o main.o shop.o
  g++ -g -Wall -o shop item.o main.o shop.o
item.o : lookup_item.cpp lookup_item.h # make .o files
  g++ -g -Wall -c lookup_item.cpp
main.o : shop.cpp cart.h
  g++ -g -Wall -c main.cpp
shop.o : shop.cpp lookup_item.h cart.h
  g++ -g -Wall -c shop.cpp
clean :
  rm -f item.o main.o shop.o
```

Python Sample

ARGPARSER TUTORIAL

```
import argparse

def main():
      # creates an argparse object
      parser = argparse.ArgumentParser()

      # specifies an argument that is expected
      parser.add_argument("square", help="display a square of a given number")

      # saves the argument from argparse object into variable args
      args = parser.parse_args()
      print args.square**2


if __name__ == “__main__”:
      main()
```