• Introduction to the Course
  - introduce ourselves to the students
  - why this course exists
    - there is a real need to teach students about the unix operating system and its spinoffs - linux and OS X
    - not enough time in the regular school year to teach you this and teach you everything else you need to know
    - and also because the CS department doesn’t have the resources to offer this course on its own
    - we needed something to do with our summer vacation
    - the goodness of our hearts, etc.
  - this course is NOT sponsored by the CS dept.
  - you will not receive any credit at all
  - who this course is targeted at
    - those students who have never worked with unix before and would to get some experience doing
    - those students who have had some experience but would like to learn more
  - the goal of this course
    - to teach you the student to teach yourself about unix
    - to give you the necessary basics to go out and teach yourself whatever you might need to know to get work done on unix
    - catch a fish for a man vs teach a man to fish
    - trying to follow Mudd philosophy - teach you to teach yourself
  - what should the student hope to get out of this course
- #pass out a course syllabus
  - should have dates and topics
- solid and basic understanding of unix
- will not teach you to program
- will not teach you emacs tricks, though we’ll provide a reference sheet
- will not solve world hunger, peace, etc
- will not teach you to be a better person, etc.

• The systems on Campus
  - discuss what turing is
  - also odin, wilkes, thuban, hopper, muddcs, math cluster
  - the filers - bambi and thumper
  - what computing resources are available to you
    - graphics lab
    - terminal room
    - AC and LAC labs
  - where printers are located
  - scanner in AC lab
  - Who Tim and Roger are...and what they do

• Philosophy and History of UNIX
  - *design and history*
  - Ken Thomson implemented the first UNIX environment
  - originally a programmers environment - because of this is extremely powerful
    - still very much a tinkerer’s environment
  - unix encourages and support “hacking” the system
- Fundamentally multi-user and multitasking
  - designed in the days of mainframe environments when many people would all work on one computer system
  - we still use the same model in many ways with turing
- multi-layered design: user | shell | programs | OS | hardware
- clean, simple interface
- scales very well - from supercomputers to microwaves
- two different flavors of unix - BSD (FreeBSD, OSX) and System V (Solaris, Linux)
- turing is running Solaris (SunOS) 5.9 - current version is 10...
- it doesn’t matter what you run, what we are going to teach you here will apply across most unixes though not all - why?
  - unix in some sense is not an operating system but rather a programming language (C)/philosophy/interface (POSIX)
  - the development of C (and the internet) are all connected
  - unix is (kind of) C, the Standard C Library, and POSIX (Portable Operating System Standard) which is the API that programmers use to interact with the system kernel
  - the design and implementation of unix also follows a certain philosophy about how an OS should be designed, how programs should be written, etc.
  - everything else is just historical baggage
  - makes unix extremely flexible - can start from scratch with a new implementation but still same “design”
  - *philosophy*
- programs should be designed to operate together
  - output of one could be used as the input to another
  - write your programs to handle text streams - universal input
- Programs are viewed as simple tools that combined will do complex tasks
  - each program should do one thing extremely well

- good analogy - LEGO blocks - simple tools but can create very complex systems

- another analogy - hammer, wrench, saw - simple tools that do their jobs very well but build structures that last lifetimes - versus a goo

- Avoid the ginormous programs that do everything
  - Windows has this problem
  - Windows programs also don't integrate very nicely - no pipeline

- want to keep everything as simple as possible
  - KISS - keep it simple stupid
  - this applies everywhere - both in OS interaction (system calls and library calls) and in programs creation

- separate mechanism from policy
  - Unix (and programs that run on Unix) provide a mechanism to do something but do not lock you down to one policy
  - policy - how a program should look and behave
  - users have to decide on policy

- principle of least surprise
  - applications will generally produce no output if successful
  - applications that fail will make it evident (sirens if necessary)
  - programs should do the least surprising thing

- there is no “one true way” - unix is not perfect!
  - unix has things wrong with it, and things that unless you understand will cause you a great deal of pain

- The Unix Hater’s Handbook - written to expose the flaws in unix

- from a stanford site:
"- The traditional command line shell interface is user hostile
  -- designed for the programmer, not the casual user.
- Commands often have cryptic names and give very little response to
tell the user what they are doing. Much use of special keyboard
characters - little typos have unexpected results.
- To use Unix well, you need to understand some of the main design
features. Its power comes from knowing how to make commands and
programs interact with each other, not just from treating each as a
fixed black box.
- Richness of utilities (over 400 standard ones) often overwhelms
novices. Documentation is short on examples and tutorials to help
you figure out how to use the many tools provided to accomplish
various kinds of tasks."
- “Unix *is* user-friendly. It’s just picky about its friends.”
- very true - have to be able to understand the system at some intuitive
  level to be able to work with it
- also if you don’t like something you can fix it, maybe not easily, but
  you can fix it

- security
  - protect users from one another and from doing dangerous things
    - protect privacy -> data
    - protect work -> programs
  - protect programs and files from users - or rather allow only
certain users to do certain things
- concept of ownership - everything on the system is owned by someone
- ownership implies responsibility - so whoever owns these files is in the end responsible for them

- Logging In
  - Need to have an account
  - two parts: username and password
  - username can be anything, though some are reserved for special purposes
  - on turing it will be the first character of your first name, followed by at most the next seven letters of your last name
  - so on turing I am: mkegel; Marshall is mpierce, Erik Shimshock is eshimsho
  - on new turing the username limit is 16 characters
  - password should be strong
    - what makes a strong password?
    - min 8 characters in length
    - special characters (Numbers and @ ! # $ % ^ & . , ( ), even spaces)
    - uppercase characters
    - should not look like a dictionary word
    - pass-phrases are good things
    - “I took CS60” could become -> “1 t00k CS 6o#” (thats a one with two zeros in took and a lower case ‘0’)
  - #Demonstrate logging in to turing
    - #notice that no password characters show up when typing
  - administrative users
    - root - is the best known account of this type - aka The SuperUser
    - owns everything on the system
    - can do anything that they want (mostly) - has to respect file perms.
- can become any user on the system
- is a standard account on every UNIX/Linux/BSD machine
- used for system administration/maintenance
  - what this is will cover later
- any other user however can be granted admin privileges, so root isn't strictly necessary to run a system

- “normal” users
  - have a home directory - directory to keep all your files to yourself
  - cannot generally see others home directories, Odin had this problem
    Charlie still does
  - can use most of the programs on the system
  - any special configuration files are generally stored in your home directory
  - on turing you have about 60 Mb for all your files
  - on new turing will have much more
  - on your own system can have as much as you want ;-)
  - the CS department does daily backups so if you lose a file you REALLY want then you can ask them to restore it

- other users: nobody, ssh, ftp, etc.
  - used to run daemons in the background
  - for security reasons do not want to have a web server running as root
  - what if someone owned your web daemon - would own your system...is bad
  - unless you’re setting up your own system don’t have to worry about this
  - and even then still don’t really have to worry about this since most of these users will have been created by default - rarely have to create a new system user

- groups
- every user on the system belongs to one or more groups
- these are sets of users which all share some characteristic
- one group, on turing at least, is students
- every student that has an account belongs to this group
- consultants are additionally members of the operator group
- however since consultants are also students, we draw another distinction
- every user has a primary group - this is a group to which file which you create will be assigned
- can generally be a member to as many groups as the admin wants you to be
- some groups serve special purposes along with the “special” users
  - can have an ftp group, www group, etc.
  - exists to allow easier management of a system by administrative users

• Introducing the shell
  - #log into turing with the default turing setup
  - what you see when starting
    - the default turing setup, easiest and most familiar setup
    - components of the prompt: username, time, command number, and directory
  - #change the prompt to something much simpler, like %
  - unix is at its heart a cmd line system - in some respects this is much more powerful than having just a graphical interface
  - it was created in the days of the teletype, like a telegraph with a keyboard only uppercase letters though - so very limited
  - so the teletype would be hooked up to the giant room filling computer
  - the teletype would send the input, and the output would be printed out on a sheet of paper, this meant that the teletype and the computer could be located
either next to each other or across the country, the computer couldn’t tell
- later the sheet of paper changed to a CRT monitor - would just print the text
to the screen
- over time the teletype changed to a full keyboard but the fundamental relation
between the teletype and computer (input and output) for most things hasn’t really
changed
- obviously today you have both upper and lower case characters
- most of you probably come from either a windows or mac world
where most things can be done with a graphical interface, for certain things
a GUI (graphical user interface) can be very good (ex. web browsing),
however, certain other things are made much simpler by the use of the cmd
line.
- for you, however, the best reason to learn the command line is that you have to
- the CS department has chosen to have students do most of their work on unix
systems, therefore to make your life easier you should learn how to use them
- other reasons to work from the cmd line...
   - simple, quick, and powerful
   - can do an enormous amount of work, with very few commands
   - for certain tasks are much superior to a graphic interface
- we will focus this course on using the command line
- becoming a unix power user involves becoming familiar with the cmd line
- #change prompt to something simple - probably use a special account on farmboy
- for now we want a simple prompt, don’t want to distract you
- when you log in you will automatically be at your home directory
- home directories on turing, and many other systems (OS X is different)
are located at /home/username
- on OSX home directories are at /Users/username
- for me: /home/mkegel (turing, farmboy) /Users/mkegel (shadow)
- directories are denoted with /, windows uses \ 
- there are other directories that we will get to later, but on turing you should never have to worry about them (you also won’t have access to them)
- there is no c:, d:, a: like on windows, everything starts at / (called root)
- file systems can generally be seen as a tree, so / is the root of the tree
- a cdrom or hard-drive can generally be located anywhere in the tree
  - no preset structure
  - working with devices we’ll get to much later so just sit tight for now
- what is the shell?
  - The shell is the highest level interaction layer between you and the OS
  - it’s what sits between the teletype and the computer - makes the computer easier to use
  - it runs whatever commands you give it, with whatever arguments you give it
  - the most important thing in learning to use unix in a productive manner is to become comfortable with using a command line
- syntax of a command
  - notation first
    - will use % to denote something happening on the command line
    (in this document at least, in the slides we’ll also use % for simplicity), such as

% pwd

/home/mkegel

- pwd was the command
- the output of the command followed
- command [switches] [arg1] [arg2] ... [argn]

- switches

  - switches are arguments that you pass into a program that modify the
    programs behavior - switch the behavior of the program
  - sometimes called flags, we’ll use the term switch though
  - different than what is generally referred to as an ‘argument’
  - a switch will generally have the form “-?” where ? is some character
  - most programs take multiple switches and also accept multiple forms for
    what is the same action
  - other style, GNU style, is “--switch”
  - windows uses the “/xyz” style - may have seen this before
  - can have multiple switches, example tar
  - tar (tape archive) is an archiving program from the days of tape drives
  - file archives are given the extension “.tar” (gee big surprise!)
  - to uncompress will most often type

% tar -xvf [tar file]

(...imagine output here...)

  - “-xvf” could have been typed “-x -v -f” but easier to write the first way
  - most programs accept many switches - gpg (file encryption) accepts 360!

- not every command takes arguments; some require them
- some basic commands to get started with - these commands are more complicated
  than what we are actually showing here, you’ll be able to find out more stuff
  about them after the end of this lecture
- these are also the commands that you’ll be using the most often
  - cd [directory] :: changes the current directory

% cd /usr/local/bin
- `pwd` :: prints the current directory you are in

  % pwd

  /home/mkegel/stuff/files...

- `mv [source] [target]` :: moves the ‘source’ file to ‘target’ file/directory

  % mv old_hw.txt not_so_old_hw.txt

- `cp [source] [target]` :: moves the ‘source’ file to the ‘target’ directory

  % cp hw_spring_05.pdf ~/trash/

- `rm [file1] [file2] ... [filen]` :: removes files

  % rm old_and_busted.wmv

- `ls` :: gives you a listing of the files in the directory

  % ls

  i_love_unix.mp3

  old_homework.txt

  resume.pdf

- `mkdir [name]` :: creates a new directory

  % mkdir hw_fall_05

  % ls

  hw_fall_05/

  i_love_unix.jpeg

  old_homework.txt
- man [command name] :: prints the “man page” for various system commands

% man cp
(...cp man page...)

- some commands are interactive
  - all these programs just run, do their thing, and then quit
  - what we generally refer to as an application is interactive
    - emacs and the shell are both interactive programs

- some things to know
  - note: unix is case sensitive - AAAA is different than aaaa - this shouldn’t be too surprising
  - also most unix commands are done in lower-case as you’ve just seen
  - most switches to programs will also be in lower case, although some need to use both - pay careful attention to both the switch you’re using and the case of the letter(s) you may be passing in
    - there are also special characters will in different contexts will be interpreted differently

- aside: some notation
  - discuss C-? and M-? notation
    - you see this everywhere, particularly in documentation, so is very useful to know
    - is incredibly simple: C == Ctrl; M = Alt (but stands for meta)
      - therefore C-x is Ctrl and lowercase c held together
    - are what might be called shortcut keys under windows
- one other notation is caret notation; ^h would mean C-h

- quitting programs

  - most programs can be quit through either Ctrl-c or Ctrl-d

  - C-d sends an EOF character

  - C-c sends an interrupt signal that causes the program to clean up
    and then to exit

  - if C-c doesn’t work and you have to quit the program, then C-\ will cause
    the program to exit without cleaning up first (as well as dumping core)

- basic shell commands

  - *tab completion*

  - wouldn’t it be nice if say you have some long
    directory name, ./thisdirectorynameistoolong/ but there was some way
    that you didn’t have to type it out every time, hmmm...

  - most shells (and any that you would normally use) provide what’s called
    tab-completion, allowing you to complete partially written statements
    into their full form

  - #this would make a good demonstration

  - so given partial data the shell will determine what sequences of characters
    will appropriately complete what you’ve written on the cmd line

  - this is accomplished by hitting the tab key

  - #hit tab key to show different completions

  - once you’ve typed enough to make the rest of the string unique (that is
    there is only one possible thing that you could be typing) the shell will
    automatically complete what you’re typing when you hit tab

  - #demonstrate on the long directory name

  - since directory names have to be unique on the same level, tab-completion
work really, really well
- but what if you want to tab complete other things?
- like say the arguments to a functions, or switches...
- luckily this is setup by default with the shell you’re using, so you can
tab complete any number of things, man pages, arguments, etc.
- *files and directories*
- up arrow gives old commands, continue pressing to work back through your
entire command history, useful for when you have to type the same commands
again and again - like
% make program
- the down arrow with of course cycle forward in your history
- *command line editing*
- the left and right arrow keys move the cursor allowing you to edit the
cmd you’ve typed on the screen - that way you can go back and correct
something without having to retype the entire command
- other useful key commands are (assuming they aren’t already used):
  - backspace (or C-h; delete on a mac) works as expected
  - delete  (or C-d; del on a mac) also works as expected
  - C-u will erase an entire line
  - C-k will erase all of the line forward from the cursor
  - C-r will do a history search #demonstrate this
  - C-a will move the cursor to the beginning of the line
  - C-e will move to the end of a line
  - M-b moves back a word, M-f forward a word
  - C-l clears the terminal
  - C-_ will undo the last thing typed
- *special characters*
  - the shell interprets a number of characters as being special
  - that is some characters have meanings to the shell other than
    what you may have typed
  - for example the semi-colon ‘;’ separates commands on the cmd line
    % cmd1; cmd2; cmd3;...
  - but what if you want to pass a special character to a program?
  - you use syntax; \\?, where ? is the special character you want to use
    - #demo echo semi; versus echo semi\;
  - so which characters are special?
    - # $ * ? [ ] ( ) = | ^ ; < > ` $ “ ‘ \
    - so when you type \; the special meaning of ; is disabled
    - you can also disable the special meaning by surrounding the character
      in quotes, either “ or ‘ (double or single quotes)
    - #demo echo “;”
    - how quoting is more powerful since all the special characters inside
      of the quotes get disabled - well mostly, there are some rules
    - pg. 527 UPT
      - ‘xxx’ - everything get disabled
      - “xxx” - everything but $ ‘ and \
    - the \ character has another meaning that you have to worry about
    - if you have a \ at the end of a line then the new line will be removed
      and the shell will treat the two lines as one long continuous line
      % echo Who \\n      stole \
the cookie, \\
from the \\
cookie jar\?
- why might you need to do this?
- you’ll find out in some little exercises we have for you

• Finding help
- our goal is not to teach you unix, but to give you the basics, then have you be able to teach yourselves what you need to know
- but sometimes you have to ask for help, and often times this is the best way to learn something new, don’t beat your head against the wall if someone you know already has the answer and can help you figure it out!
- so part of teaching yourself is knowing when to ask questions
- there is one resource that is the best and it is? ...Google!
- whenever you have a question about something the first thing that you should always do is google for the answer - most likely someone else has already taken the time to solve and write up whatever solution you are looking for
- lots of very good resources
  - websites (particularly forums), books, documentation, and people
  - some good websites to check out are: linuxdoc.org, freebsd.org
  - some good books are: UNIX Power Tools, FreeBSD Handbook...
  - people: Marshall, Mark, Mac, and many others!!
- Computer Science Department Staff and Consultants
  - they have people hired to help you out!! You should use them as a resource!
  - Staff take care of the various boxes the department runs
    - #give a listing of the boxes, their purpose, and their addresses
- turing.cs.hmc.edu -- shell server
- #find new mail server
- muddcs.hmc.edu -- web

- Consultants sit in the terminal room and answer Unix and CS related questions
  - #give a list of who are the consultants, emails, and schedule

- The CS department website
  - QREF’s - documentation written specifically for YOU
  - is a great resource for general information, system policy, etc.
  - #give the URL for the QREF’s
    http://www.cs.hmc.edu/qref/

- the professors will also help you out as much as they can

- some good mailing addresses for questions
  - linux-l@cs.hmc.edu - linux mailing list for hmc students and others
  - #give out others (?), maybe forums.muddstudents.org

- man pages
  - purpose is to document how each of the various programs/commands on the system work, and how to get them to do what you want
  - generally short on examples, but in general well worth reading
  - can document other things
  - get in the habit of using man pages - saves time and effort!

- what man pages are not
  - general reference, programming guides

- what man pages are
  - very specific references on a limited set of system commands
  - essentially they document how to use the programs on the system
- some man pages break these rules but in general they seem to be followed

- Other systems

- info

  - is mainly about emacs

- the ‘help’ command for bash

- Most programs come with more documentation than just a man page

- check out: /usr/doc and /usr/share/doc

- slackware uses /usr/doc and is really good

• How to use man

  - how man is organized - sections - explain what each section covers

    - 1 - Commands available to users

    - 2 - Unix and C system calls

    - 3 - C library routines for C programs

    - 4 - Special file names (Devices and Device Drivers)

    - 5 - File formats, protocols, and conventions for files used by Unix

    - 6 - Games

    - 7 - Conventions, Macro packages, Word processing packages and Misc.

    - 8 - System administration commands and procedures

  - always need to keep in mind that one command could be in two+

different sections at the same time

  - allows for a better organization of documentation

- some switches

  % man -k perl

  (show output)

  - so then “man -k” allows you to search for man pages
- “-s” - specify the section to search

- “-a” - show all man pages that match the search, not just the first

% man -a sync   (demonstrates this)

(...on the mac shows two pages...)

- how to read man pages - the parts of a man page

- at the very top is the is the name and section number
  for man it is man(1)

- NAME - just a quick one line description

- SYNOPSIS - all the arguments that the program accepts

  #-how to tell if something is optional or required

  - optional things are surrounding in brackets ‘[xxx]’

  - required things are not

  - #show example - man requires at least one argument

  % man man

- DESCRIPTION - a paragraph “summary” of the program

- OPTIONS - what each of the switches to the program do

- ENVIRONMENT - what environment variables the program expects
  (something we’ll cover later on)

- SEE ALSO - other related man pages to see

- BUGS - bugs the program may have

- FILES - files that the program may use (usually configuration files)

- AUTHOR - who wrote the program

- navigating a man page - or rather learning to use less

  - arrows move man page up/down left/right

  - space bar moves a screen at a time

  - to search would type /pattern, then press enter
- demo this feature
- typing / or n again moves to the next instance of pattern
- N moves to the previous instance
- b: back - like page up
- f: forward - page down
- man almost always uses the less program as its pager, so learn to use less!
- if man is not using less, then it is using an older pager called more
  which is much worse, should fix if encountered
- exercise - now for a man page hunt

- Making programs work together on the cmd line
  - you now know how to run programs on the cmd line
    - you just type them in!!
    - and what some of the issues are with special characters (quoting)
  - but unix is about more than just running one program at a time (that’s windows job); in unix programs work together, so we’ll show you how to do that
  - say you want to know the number of users on the system at any given time
    - could have one program to keep track of this - would have to keep track of users and then print that number
    - or could have a program that tells you who is on the system - could output that to the screen, one line per user
    - and another program that counts lines
    - combine the two and you have the one more powerful program
  - unix is about combining simple programs
  - you should already know about stdin, stdout and stderr from CS70 or CS60
  - they allow a single program to get input from the cmd line or print output
to the cmd line

- thats the basic idea

- a bit of review though

- stdin - represents the characters you are inputing

- stdout - the standard place for the program to print output

  - general goes to the terminal you ran the program from

- stderr - like stdout but for error output

- in C++ the three are accessed through cout, cin, and cerr

- can accessed in other ways through other languages

- #ask question: so we have some idea of what stdin and stdout are, they represent the input we get from the terminal and the output we send to the screen, but how do you think that they actually work?

- we need one idea: everything in unix is a file - this is an important idea also
  
  - devices (keyboard, mice, monitor)
  
  - directories (yep even directories)
  
  - links (we’ll cover what these are)

  - and what you normally think of as files

- so stdin, stdout and stderr are files? yes!

- why so surprising? you read() from stdin, write() to stdout & stderr

- so you are just reading and writing to and from special files

  - the OS takes cares of these “files”, no need to worry about deleting them

  - each program has its own set stdout, stdin and stderr that is managed by the Operating system

- how do you combine simple programs? how do you get two programs to talk?

(good time to use the chalk board)

  - #ask group for ideas
- we know that two programs can share the same file - can have the same file open two different times
- so to communicate one program could write to a file, the other could read
- [program one] --> [file] --> [program two]
- can use the stdin, stdout, and stderr files to have programs talk to each other
- [program one] --> [stdout/stdin] --> [program two]
- the stdout of one program IS the stdin of another
- very flexible and powerful solution
- instead of the program printing its output to the screen it is piped to the second program
- keyword: pipe
- you can do this from the shell
- the syntax is
  
  % cmd1 | cmd2
  
- the ‘|’ character is known as the pipe and is located on the \ key, so Shft-\ is |
- example of this in action - first run who and tell what wc does, then run this
  
  % who | wc -l
  
  (gives the number of terminals open on the system)
- #get a longer piping example
- programs used in this way are called ‘filters’
- can do more than just piping between programs
- you can redirect the output of a program to a file
- you can redirect a program to read stdin from a file
- keyword: redirection
- you can have whatever file you want to be stdin, or stdout/stderr,
using the shell
- redirect file to stdin
% cmd1 < file1
- You use the < to have file be stdin for the program
- redirect stdout to file
% cmd2 > file2
- So you use > to redirect the output
- to send stderr to a file do
% cmd3 2 > file3
- Some notes on redirecting output
  - the file cannot already exist (if you have NOCLOBBER set)
  - won’t execute if it does
  - since this is set by default on turing you need to be aware of it
- *Appending a file*
- You can append to the end of a file (add lines of standard output at the end without writing over what is already there, useful for log files) by doing the following
% cmd4 >> file
- #have a demo ready to append
% echo “Hello there!” > newfile
% echo “Good to see you.” >> newfile
% cat newfile
- can also append stderr to a file - better for log files since you don’t generally want to send error messages along a pipeline, only want real output
- to append stderr very much like redirecting stderr
% cmd5 2 >> file
- will want a few complex examples of piping and redirection
- # 3>&2 2>&1 1>&3 to swap err and out
- not every program takes things from stdin or send output to stdout
- for example can’t do
  
  % emacs < myfile
  
  - instead have to send myfile in as an argument to emacs

- Another way to combine programs
  
  - you may remember the ` as being a special character
  - ` is located on the key above tab and left of 1
  - is called a back-tick or back-quote
  - anytime you surround a string with back-ticks the shell will treat it as a command and then replace the string with the output of the program
  
  % kill -HUP `cat /var/run/sshd.pid`
  
  - this will restart your ssh server, since `cat..` returns the PID of sshd
  - another example
  
  % emacs `grep -l error *.c`
  
  - this will let you edit all of the files that have the word error in them
  - another
  
  % pine `users`
  
  - would allow you to send an email to everyone logged in to the system
  - lots of other uses, mainly in shell scripting

- Processes and the Kernel - Actually running programs
  
  - now that have some basic tools lets do something with them
  
  - review of tools
- piping, redirection, stdin, stdout, basic commands, how to find help
- should also be comfortable with using the shell
- some of the commands we’ve been running haven’t actually been “programs” in the sense normally described
  - example: cd is a builtin, its a part of the shell and not a stand alone program like say emacs is
- need to have some sense of the basic, underlying abstractions of how the computer works - not on a bits and byte levels, but higher up
- again is a layer thing - bottom up
  - hardware - the machine itself - just electrical signals
  - kernel - a mediator between the hardware and the user - provides a set (library) of functions for controlling the hardware that every program you would write would need (ex. drawing text to the screen, writing to disk)
  - applications - do the things that you want the computer to do
  - the user - running applications and making sense of the electrical signals that the computer processes - only here do things have any “meaning”
- in understanding unix you don’t so much about the hardware - leave that to the engineers
- really only care about the applications layer - how to make programs do what you want
- but in order to fully appreciate and use unix you have to understand at some level how applications interact with the kernel, and what a kernel is
- first though...
  - #how is a program different from a process?
  - #how many copies of each do you normally have?
    - 1 for executable
- n for processes

- #why would you have more than one process of a single program?

- need them to be doing different things - two xterm windows for example

- #ask what a process is

- a program in action

- an abstraction provided by the kernel

- every process thinks its the only one running - runs in a self-contained environment with the kernel being responsible for enforcing this abstraction

- #ask what a program/executable is - just the file containing instructions

- #is very broad: does it matter what kind of instructions?

- the instruction could be machine code

- could be byte code like in java or compiled python

- or could be an interpreted scripting language

- #how do the processes get run?

- the job of the kernel is to manage the many processes on the machine

and run them in some sane order so that the system is both responsive

and productive

- the kernel also provides an interface to all the hardware on the system

- this interface is uniform across architectures and hardware

- so you always call read() though the disk might actually be

just a network share somewhere in asia

- #does the kernel just run processes? basically, and do what processes need done

- #list the other jobs of the kernel

- manage devices

- reading/writing files - handling system calls

- #the kernel does a bit more
- enforcing system policies - security for example
- allows for multiple users on a single system
- kernel does other things also
- take Operating Systems if you want to write a kernel
- take CS105 to learn more about how Computer Systems work
- as users all we care about is running programs, leave the other stuff to the CS geeks
- unix is very good at doing this
  - unix is a multitasking environment
    - can run multiple processes at the same time
    - no fear of them interacting
    - each process runs in its own little environment isolated from others
    - unless you WANT them to interact! very key!
  - unix is a multiuser environment
    - can handle multiple users using the system at the same time
- so does all of this have to do with you learning to use unix - not a whole lot other than making you understand that you can run multiple programs with fear of them crashing the system like on some other systems we could mention
- so if you are going to run multiple programs from the shell(s) it would be helpful if you knew how to manage them
- with GUIs most of these things are not really an issue, but its still very useful to have some idea in case you mess things up
- can open as many xterms as you want, but at some level you need to know this...
- from a single shell session can run about as many programs as you would ever want to - constrained by memory and processing speed
- normally though you are only ever running one program at once in a shell;
this is referred to as the foreground process

- foreground process is the one that receives keyboard input (if interactive) and prints to the terminal

% mv ..
% ls ..
% rm ..

- sometimes you want to other programs doing things also

- say stop reading a man page to copy a file, or read email
  - could open another terminal - do you really want to do this for every program? what if you only have a limited number of terminals?
  - or could “suspend” the process and put it in the “background”

- managing processes like this is called “job control”

- recall that C-c kills the program you are currently running

- sometimes its C-d (sends an EOF -- usually for interactive programs)

- to suspend a running process use C-z

  - common mistake to think that the program has just suddenly died
  - people unfamiliar with the system will just start over, being angry and frustrated at the work they “lost”

- I’ve seen this happen to people two or three times...

- speaking of other frustrating things also mention the C-q and C-s control flow “bug”

  - C-q     resume console output
  - C-s     stop console output

- #demo reading “man page”, suspending, and using pine

  - will want to demonstrate that pine can not be suspended by default

- just because a process is suspended does not mean it isn’t running
you could have a copy operation running in the background

music program in the background

also note that the system has to run all these, so no free lunch!

remember: limited by RAM, hard drive speed (swapping), and cpu power

to switch back to the process you just suspended you use the fg command

can of course suspend multiple processes

#demo two other man pages, also suspend

#will have three jobs in the background with cmd line open

when we type

% cmd

this program is the one we’ll be running in the foreground

to run a program in the background we type

% cmd &

this causes the program to run, but not in the foreground - so it won’t be receiving any input that you type unless you switch to that program

#open emacs in background and switch to it

% emacs &

% do some thing on the cmd line

% fg

this is one of the most useful things to know, so remember it!

why this is useful - give this example scenario

are logged in remotely with no X session

have an emacs window open and change a bit of code

too slow to exit emacs and restart

so press C-z to suspend

then can rebuild from the cmd line
- then can switch back to emacs to continue working

- really only need to remember C-z to suspend and fg to bring the process back

- *Managing processes*

- use the jobs command to list all the jobs that you have running

% jobs

[1]  + Suspended  man page
[2]    Suspended  man perl
[3]  - Suspended  man printf

- the number identifies each job

- we use the fg command to switch back to which ever job we want using the number
  - by default fg will bring back the job with the plus sign
    - this is the current job
  - the minus sign was the job that used to be the current job

- the rest of the output of jobs is obvious
  - Stopped means it doesn’t receive any cpu time
  - running means just that

- example to go back to man perl we would type

% fg %2

- nifty trick: if we know the name of the process, say we have a single emacs process along with other can quickly switch back to emacs by using a pattern matching scheme for fg

- instead of %n, you have &?xxxx where xxxx is the first part of the name of the running process
  - obviously you need enough of the name to disambiguate the process

- so returning to emacs could be as simple as typing

% fg %?em
- don’t have to use jobs and look there, just do a single fg command
- now that we can manage a few processes what about the processes on the rest of the machine?

- Understanding process ownership
  - every process on the machine has a parent/child status
  - so when you log in and get a shell that process will act as the parent to all the other processes you create on the machine
  - there is one master process on the machine, init
  - every process is in some way a child process of init
  - #draw out a parent/child diagram from init down to the shell
    - init -> sshd -> tcsh
  - processes are also owned by users
  - each user on the system can own processes, so when you log in the shell that you see is owned by you
  - to identify all the processes on the machine each process is given a PID or process id that uniquely identifies the process
  - you use the ps command to see the processes that are running
  - run just by itself (no switches) will just print out your process that are being run from the terminal you logged in on
  - to see everyone’s processes you would type (on turing)
    % ps -ef
  - everywhere else (linux, bsd, osx) you type
    % ps -aux
  - can see what everyone is running with this
  - #point out which column is the process id
- pids are not always 5 digits
- init for example will always have its PID equal to 1
- on OS X pids are generally 3 or 4 digits
- linux tends to be 5 digits
- solaris, which turing is currently running is also 5 digits
  - will always between 2 and 32000 (signed word)

- so why is this useful?
- what if we want to kill a runaway process or any other process
- imagine you have a java application in an infinite loop
  - #this would be a good time for CS60 students to pay attention
- don’t want it running forever, so want to kill that process
- we use the kill cmd to do this
% kill [PID | %n]
- two ways to kill
  - (1) if the process is running in the background, can use background number
  - (2) can use the process id (PID)
- we can use ps to find the process id
- #demonstrate using ps to find the id and kill the process
- but how did we know that the process was out-of-control in the first place?
- nifty utility on unix called top
- will list out the processes in order of cpu time
- #demonstrate top - point out the PID column
- can use this to see if any of your processes are out of control
- on turing an out-of-control process will generally be taking up a full cpu,
  which you can see by the fact that the process would be consistently using 16%
- #should be able to point out several aspects of top
- memory usage, cpu time, state and priority

- top is made more useful by seeing only the processes that you own

- first start top

  % top

- then press the ‘u’ key; then type your username, or another username if you want to see their processes

- then enter

  #demo this feature also

- to get help on the other commands top offers use the ‘h’ key when running top

- to quit top press the ‘q’ key or C-c

- top is really nifty, so spend a few minutes trying to figure it out

- digression: what would happen if say, we were root and we typed

  % kill 1

- what would happen? is that even referring to a process?

- yes it is - init (or launchd on Tiger) is given PID of 1

- so typing kill 1 while root would kill you entire system - basically all processes stop and the system has to be rebooted

- this is a bad thing and should be avoided

- *process priority*

- #what is priority?

- priority defines how much time the particular process should get from the machine

- processes with a low priority gets run less often than a process with a high priority

- as a user you can’t actually change the priority, only root and the scheduler can do that

- what you can do is set the *niceness* of the process
- A higher nice value means that the process is nicer to other processes on the system, giving them more time to get stuff done.

- So a high nice value leads to a low priority and a low nice value leads to a “normal” priority.

- To set the niceness of a process you use the nice cmd.

  `% nice cmd`

- If you are going to be running a long and fairly intensive job on Turing, please use, and be, nice!

- Nothing worse than locking the system up for fifty people while you try to compute prime numbers ;-)!

  - #show how mprime has been niced on Odin

- To use nice run it like such.

  `% nice cmd args...`

- #find a demo for nice

- Check out the man page for more info.

- Does nice have an effect?

- Nice will have an effect if the process is *CPU bound* meaning that it requires the processor more than anything, doesn’t really need any input.

- On *IO bound* processes not so much since they wait for input anyway, and while waiting the system will do other things, won’t have much effect.

- Now know all about managing processes and how the system handles them.

- Can now show some useful programs.

- Some programs to remember.

  - #this is a list of programs that we consider to be the most useful or that are basic to being able to use a Unix system.
- will need to have demos of all of these commands

- *locate, echo, cat, clear, less, head, tail*

- locate

  - allows you to find files on the system
  - has to search a database so not always up-to-date
  - usage

% locate <filename>

(listing of files)

- accepts wildcards for files which we’ll get to later but which you may already be familiar with...

- echo

  - prints whatever arguments that you give it to standard output
  - useful for when you want to write/append some string to a file
  - or just when you want to write to stdout
  - usage

% echo “text string”

  -or-

% echo text string and more text

  - before using you should really read the man page

- cat

  - like echo, instead takes file names as arguments
  - prints the contexts of each file in order to standard out
  - one of the more useful commands, especially if you want to look at a small text file
  - cat is short for concatenate – you may think they should have used “cons”
  - usage
% cat <file1> <file2>

(contents of files)

- clear
  - clears the screen - useful for when your terminal has filled up with junk and you’d like to make it go away
  % clear

- less
  - is an application that allows you to view a text file and page through it that is, look at whatever parts you want to in whatever order - aka pager
  - have already had exposure to this program through the man command
  - man opens up its man pages in less (not by default but generally)
  - but of course can use less to view more than just man pages

- usage

% less <file>

- and some of the key commands to make less do what you want - interactive
  - arrows move man page up/down left/right
  - space bar moves a screen at a time
  - to search would type /pattern, then press enter
  - #demo this feature
  - typing / or n again moves to the next instance of pattern
  - N moves to the previous instance
  - b : back - like page up
  - f : forward - page down

- head
  - prints out the first n lines of a file to stdout
  - useful for when you have a sorted list and want only the first n
- by default (if n is not specified) will print out the first 10

- usage

% head [-n XX] <file>

- tail

- like head, in that it will print out the last n lines instead of the first
- more useful in that files, especially log files, are often appended to
- can then follow these files for as long as they are open and see what
is being added to them

- usage

% tail [-n XX] [-f] <file>

- #now on to some other commands

- the next two are both searching commands - one is for searching through
stdin, the other for searching through a directory tree

- grep

- allows you to search for a regular expression within a file
- often though you just want to find a string within a file, not every
thing matching a certain pattern
- this is really just a copout since we don’t want to show you regex’s
until later

- so to search through a single file

% grep [switches] “quoted string” file1 file2 ... fileN

(show output)

- grep has some useful switches that make it nicer to work with

- “-c” : instead of printing lines that match will instead count the number
of lines that match and print that instead

- “-n” : print the line number on which the pattern was found
- “-H” : prints which file the pattern was found in
- “-r” : if a directory name is given will recursively descend through the hierarchy looking for matches in all the files in that tree
- there are also different versions of grep, so you’ll want to check out the grep man page, as usual

- find
  - at its most basic level find just descends through a directory tree and prints out the names of whatever files it finds there
  - so you can replicate the locate command with
    \% find ./ | grep pattern
  - this is generally much slower than locate, but has to go through the entire file system starting at ./ (which is where ever you currently are)
  - but find is much more complex than this and can do locate all on its own
    \% find ./ -name glob -print
  - the “-name” switch will search for the glob, and “-print” will print it to standard out (note that glob != regex)
  - find can of course do much more interesting things
    - try to find out what this does, and why you would need it
      \% find ./ -name “.foo” -perm +444 -exec chmod -R a+r {} \;
- these are other commands which are useful, but not exactly necessary to know
- w
  - prints out who is on the system
    \% w
  
  (show output)
- which
  - prints out which command exactly will be executed when you just type cmd
- useful for when you want to figure out either which program you are using
or where the program you are using is
% which cmd
/some/directory/cmd

- du

- displays the amount of space that a file (or directory) is taking up
  on disk
% du file1 file2 file3 ...
(show output)
- to get the output to be a bit more useful, will want the “-h” switch
- #demonstrate this
- if file is actually a directory, du will recurse through the directory
  printing out the size of each file as its gets them finally printing
  out how large the directory is in total at the end
- this is not very useful, so pass in the “-s” switch to suppress the
  extra output
- this creates a useful “sizeof” command
- du -sh file1 file2 file3 ...

- df

- shows a detailed listing of the amount of space being used on each
  file system that is mounted
- #show the output of and explain
- made nicer by the “-h” option
- #show this output
- #at the end of this would be a good time to poll to figure out what other
  programs people would like to know about;
• Files, programs and file, and the filesystem

  - what is a file? some review from before

  - a file is a stream of bytes

  - traditionally think of a file as something like a text file or mp3

  - in unix though we know that everything is a file: directories, devices, etc.

  - review the unix security model

    - users have a home directory

    - users not allowed to do random (dangerous) things - only allowed to do those things that they have permission to do

    - there is a superuser that can do anything

    - users belong to groups, and have a primary group

  - files have their own security model

  - files have an owner and a group that they belong to

  - user is you

  - group is the group the file belongs to

  - so when you create a file, the file is created with you as the owner, your primary group as the group the file belongs to, and all the permissions on

  - #do an ls -l on some files

  - #will want to draw most of this out on the board

  - #demonstrate the two following programs

    - chgrp - group

    - chown - owner

  - so we’ve defined two categories that a file belongs to, what else is left?

  - what about the other people on the system?
- so in all a file is concerned with three sets of people: user, group, others
- others are by definition everyone else on the system
- besides owner, group, etc., files have permissions associated with what people inside of these classifications can do with the file
- so just because you own the file doesn’t necessarily mean that you can write to it, or execute it if it’s a shell script
- so what things can you do to a file?
- well unix defines only three (there are more that could be useful, but unix is old and the spec. hasn’t changed with the times)
- these three are: read, write and execute
- can set these three permissions for any categories: user, group, others
- so the permissions look like: rwx|rwx|rwx
- the first rwx is the user, the second group and the third others
- #have an exercise to figure out what the permissions of several files mean
- #so when you see rw-rw-rw what does this mean? etc.
- now on to octal permissions
- they are permissions base 8, because each permission group (user, group, others) can be represented by a number from 0 to 7
- if we write out 0 to 7 in binary we see that we require 3 bits (hmmm!)
- _ _ _ so these bits have value 4 2 1 and are mapped to r w x
- r w x } draw out something like this
- 4 2 1
- so 7 looks like 1 1 1
- and 4 looks like 1 0 0
- combining this we can write permissions out as a group of three numbers 0 to 7
- permissions look like xxx
- the first X is user, the second is group, and the third is others

- examples of common permissions settings
  - 666: everyone read, write
  - 777: do anything to file
  - 022: user can’t do anything, group w, other w
  - 744: user do everything, group r, other r

- #as an exercise write out the octal permissions in the rwxrwxrwx form

- *changing permissions*

- how do you change the permissions of a file?

- so with chgrp and chown we could change the owner/group of the file

- what is the equivalent program with permissions?

- chmod - will also change ALL aspects of a file if wanted

- #demonstrate some standard examples - make file world readable

- switches u, g, o and the +-= syntax, with perms r,w,x

- also cover -R for recursive

  - since may want to change all files in a directory

- *special types of files*

- directories

  - directories are files too, though can’t read/write to it as you would
  a regular file

  - but because are files, they have to have the same permissions that other
  files on the system has

  - unfortunately the permissions on a directory act a bit differently (which
  you might expect)

  - #list how permissions affect a directory file

  - r can list with cd
- x can access the file within
- w means can

- dot-files

- so when you type the ls cmd, do you see every file?
- obviously not since I’m asking this question
- #do an example of a hidden file - .hidden or “.you don’t see me”
- #is that all the files in the directory?
- #not necessarily...

% ls -a

- “-a” switch will show ALL files in the directory
- this is useful because dot-files are generally used to hide configuration information for programs away from the user
- also keeps the configuration information in your home directory so that each user can have their own settings as they like

- some examples of .files
- one common one you may care about is your .zshrc, this controls some of the settings you have in your shell as you login
- another is .emacs - controls the settings for emacs
- you can also have .directories
- these are directories that are hidden
- do the same thing as .files (hide config info), but for programs that may have many configuration files that need to be hidden
- so is it a nice thing that just ls doesn’t show all the files in a directory by default? yes
- #show a gftp listing of my home on farmboy

- some really special files
- so when you do a `ls -a`, do you recall seeing two files one named “.” and the other “..”?
- probably not but these are two special files
- the file named . is this directory
  - #do a literal translation for whatever directory in on board
  - so if you type
    
    % cd .
  - you stay in the same place
  - what if you see a dot in the full path name of a file?
  - ex. /foo/bar/./baz/box/
  - where is this referring?
  - to /foo/bar/baz/box/
  - so it means nothing, should just ignore it
  - ./ is the same thing as .
- the file named .. is the parent directory
  - what about the other file? “..”?
  - you should already know that if you want to go up in the directory hierarchy that you type
  - therefore .. is the parent directory of the current directory
  - ../ is the same as ..
    
    % cd ..
  - so can probably guess that is you see
    
    /foo/bar/../baz/box/
  - you know where this refers to? where does this go?
  - answer: /foo/baz/box/
- so you can't ignore these, but can be useful for when you need
to craft a path to a file

- *other special files*
- tilde (~) denotes your home directory

% cd ~

- will take you to your home directory, a plain cd will do the same
- ~username denotes the home directory of that user

- *relative and absolute paths*
- can refer to files in two different ways
- what if you have written a program and want to access a certain file - config.cfg
  - #how do you refer to this file?
  - two ways: relative and absolute path
  - absolute paths never change and describe in “absolute” terms where a file is in the tree

% emacs /home/mkegel/school/fall05/graphics.txt
- absolute paths will always start with /
- you are starting at the root and working down the tree
- relative paths start with a ./ - the directory tree before the ./ is not known
- don’t necessarily know where the program will be stored, but always know where the config.cfg file will be in relation to the program
- or should at least know where it will be
- can refer to it then by saying

% cmd ./one/two/config.cfg
- the ./ is filled in by the OS or the shell - depending on context

• Links - symbolic and hardlinks

- so now that we’ve confused you this much what if you wanted to have two files
have the same name?

- why would you want this?

- how about when you upgrade software - could have the program named emacs, but the /real/ name might be emacs-21.2, an older version could be emacs-20.7

- you want to continue to access this program through the same name, but have it always refer to the new version

- #how might you do this?

- unix uses a thing called links

- links are a special type of file which transparently redirect you to a different file

- so you can type emacs, but the OS will know that you really mean emacs-21.2

- can link both directories and regular files

% ln target link_name

- the basic syntax is backwards, so to “link” emacs to emacs-21.2 you would type

% ln emacs-21.2 emacs

- there are two types of links: soft and hard links

- a hard link, which was what we created above, is literally a different name for the same file

- the new name is stored in the directory its supposed to be in so takes up virtually no memory

- the hard link literally points to the same data on disk

- a hard link cannot cross file-systems (say you have two different hard drives and want to have one file refer to another)

- and you cannot hard link to a directory

- a soft-link is different

- it is a separate file (not just an entry in a directory file) that the OS
interprets as the real file, basically just a type of pointer

- soft-links can go across file systems and can point to directories

- in practice you should probably almost always create soft-links

- to create one pass the “-s” switch to ln

  \% ln -s emacs-21.2 emacs

  - this will create a soft-link

  - #show output of ls -l

• File globbing

  - many times will want to work with multiple files

    - say you want to mv or cp or rm all the files in a directory

    - how do you do that?

  - you use a thing called file globbing

  - is very similar to UNIX regular expressions, but should not confuse them

  - globbing is provided by the shell

  - this is key - is one of the many complaints the people have about unix

  - the shell is responsible for expanding globs

  - programs generally only take file names and don’t generally do pattern matching

  - some people see this as a bad thing - for you you don’t have to care

  - some standard globs - see page 658 U.P.T.

    - * - matches all files in a directory (not dot-files though)

    - ? - match one character

    - [a-z] - any character a to z

    - (x|y) - match either x or y - this is not pipe!

    - #give some examples and ask what they would match

      - a[a-z]*
- ?[c]*c
- can put these anywhere in the expression
- by far the most common is to do
  - ex1*ex2
  - *ex3
  - ex4*
- to match dot-files have to put a dot explicitly
  - to match all dot-files in a directory would do .*
  - #show listing of ls .*
  - note that this also lists “.” and “..”
  - so what would happen say if we did this...
% chmod -R a+r ./
- as root from /home?
- much badness is what...since more than just the down directory will be changed
- .. follows to its parent and so you’ll also change all the files in all the directories at the same level and below in the tree
- draw out to illustrate
  - #have an exercise of having to mv some set of files from one directory to another that match some specific pattern
  - #more similar exercises could follow

- The tree of life - How the Unix directory tree is organized
  - now that you understand how to work with files, how to work with programs should know a little bit more about the environment you are working in
  - unix is in some way a very structured environment
- 35 years of tradition have specified how you do certain things
and why they are the right things to do
- and then we come to the way the directory structure is organized
- what do I mean by this?
- so with windows you were probably comfortable having all your programs be
installed to c:\program files\program; so everything you installed was in a
common place
- what if some one decided one day to instead install everything to
c:\Your Programs\Program and c:\My Programs\Program; but all the old ones
you had would (by default) install to the old directory
- this would be quite a pain
- in unix the same thing exists, but across the different unixes that exist
- of course unix is supposed to be independent of the directory structure on
which its working, just makes it a pain for people in real life
- this is an historical artifact
- in order to gain market share one thing they teach you in business school
is to differentiate your product
- so when unix began to come out in various commercial flavors the companies
changed things a bit to reflect how their product was different
- the most obvious and easy way to change an OS is to change where you put things
- so you end up with apache (a web daemon) being in four different places
in four different flavors of unix
- luckily the madness has continued!!!!!
- linux suffers from the same problem though perhaps not to the same extent
- where things are put in debian is different from gentoo, is different from...
- FreeBSD has some resemblance to the linux directory structure similar
just to limit the confusion we’ll just go over the basics which apply to
most unix systems, except for OS X which is very different

- *now the real material*

- #draw a /

- so we know that everything in unix starts at /

- so how do we decide what goes under this?

- we don’t; luckily has already been done for us

- so lets start with something you already know /home

- */home*

- /home is where the ♡ is

- #cd into each directory as they are being discussed

- in some sense this is true since this is where the home directories
  of all the users on the system are stored

- the home directories generally have the form /home/username, though this is
  strictly necessary, just useful

- */bin*

- rest of this is taken from

% man hier  (on OSX, also present on FreeBSD, haven’t seen it on Linux)

- #see also pg 822 UPT

- /bin/ user utilities fundamental to both single-user and multi-
  user environments

- /dev/ block and character device files
  fd/ file descriptor files; see fd(4)

- /etc/ system configuration files and scripts
  X11/ configuration information for X

- /sbin/ system programs and administration utilities fundamental to
both single-user and multi-user environments

- /tmp/ temporary files
- /usr/ contains the majority of user utilities and applications
  - bin/ common utilities, programming tools, and applications
  - games/ Unix games, also fortune
  - include/ standard C include files
  - lib/ archive libraries
  - libexec/ system daemons & system utilities (executed by other programs)
  - local/ executables, libraries, etc. not included by the basic operating system
  - sbin/ system daemons & system utilities (executed by users)
  - share/ architecture-independent data files
    - man/ manual pages
  - src/ storage for kernel/other source code under linux
- /var/ multi-purpose log, temporary, transient, and spool files
  - log/ misc. system log files
  - /lib - shared libraries and kernel modules
  - /opt - larger static packages - where you would install KDE for example
  - /boot - kernel image and other boot files
    - this is where the linux kernel gets put
  - /mnt - temporary/permanent mount point for devices
    - devices can be mounted anywhere, so one solution, if you have lots of
hard drives, optical drives, cameras, scanners, etc. is to make lots of
mount points under /mnt, so /mnt/dvd/, /mnt/ipod, etc.
- is this a good thing?
- all up to your own preferences
- /mnt is meant for (at least in FreeBSD) a single device that won’t
be mounted for any length of time
- would mount to a different point, say /ipod, so mounted right from root
- are multiple solutions, could also have a /vol (/Volumes)
- you now have an excellent understanding of the basics of UNIX

only a little bit left to look at

• Some special devices
  - /dev/null
    - this is a sink for input, you can send anything here and it will
      just go into the void
    - useful for when you need a place to get rid of output, but don’t want
to waste space on the disk
    - if read from will give you an EOF
  - /dev/zero
    - produces an infinite stream of zeros if read from, good for when you need
to null something out
    - if written to will just give a permission error (?? Is this always true ??)
  - /dev/random
    - when read produces a random stream of bytes
    - permission error when try to write to
• More on /etc
  - *other config files stored in /etc*
    - NFS: exports
    - system: group, passwd, and shadow
    - shell: profile
    - sudo: sudoers
    - network and inet: hosts, hosts.allow, hosts.deny, hosts.equiv
    - lots of other stuff

• Environment Variables and Shell Variables
  - these are special variables that all programs know about
  - they exist in the background and let programs know about certain things
    - where to find programs to run, which editor to use, etc
    - serves the same purpose as the registry (sort of) in windows
  - #cover the distinction between the two
    - environment variables are inherited across all shells you may have
      and all programs that you run
    - shell variables are local to each instance of a shell that you may have
    - each shell gets a clean slate of shell variables when it starts
    - analogy - programming: global versus local scope
  - variable naming convention
    - CAPS are environment
    - lower-case are shell
  - please follow convention!!!
  - you generally only ever care about environment variables
  - #show listings of the variables
- how do you see what environment variables you have?
  - simple use the printenv command

% printenv

- what about shell variables?

% set (lists everything: functions, shell, and env. variables)

- #pg 705 has a good listing of pre-defined environment variables

- good ones to know about - will be all caps
  - PATH - when you type in a command these are the directories that
    the shell will search for the cmd you have typed
  - HOME - where your home directory is located
  - HOSTNAME - name of the system you are on
  - SHELL - full path name of shell you are using
  - USER - who you are logged in as
  - UID - your user id
  - TERM - type of terminal you are using

- #so how do we access and set these variables?

• The Different Shells
  - now is a good time to talk about what shell you are using
  - there are in fact many different shells!
  - you’ve probably been using tcsh, the turing default, all this time
  with ever knowing it
  - there are a multitude of shells, each with different strengths and features
  - most fall into two different categories
    - those based on sh, which was the first shell
    - those based on csh
# give proper pronunciation lesson

- the default on turing is tcsh, which is based on csh
- other shells include bash (probably the most widely used shell), ksh, zsh, ash
- probably the best (certainly the most fully featured) shell is zsh
- we recommend that you use either zsh or, if you have to, bash

- why use zsh?
  - its just plain nifty
  - will tab complete many very cool things, and even not so cool ones
  - e.g can tab complete kill and man for example
  - also we are only going to demonstrate for this shell, you have the tools
to figure out how to do the same things for yourself on your own shell

- so how are they different? what really distinguishes a shell?
- in terms of user interface, shells are almost identical
- shells provide the ability to run more one command at a time
- if you string commands together in a file, with some control structures
like if, while, and case statements: you get a language
- called shell scripting
- fundamental different between csh and sh - csh is more based on C syntax
- sh is just funky and weird syntax, but for shell scripting is better
- so when we say a shell is based on another, what we really mean is that
zsh or bash can run scripts written for sh
- and that zsh and bash offer scripting options that weren’t available with sh
  - the same for tcsh
- we won’t show you shell scripting yet, but you will learn about it

• Setting Environment Variables
- to set environment variables in zsh
% export NAME=value
- can also do this
% NAME=value; export NAME
- you can see the output of a variable by using the echo cmd
% echo $ENV
- have to remember the $
- #demonstrate this for the students
- the most common thing to do is to add paths to PATH
  - may have to do this for a class
  - may want to do this depending on which box you are logged into
- couple of ways to do this
  - (1) from the cmd line
  - (2) when you log in - as part of shell configuration
- from the cmd line for zsh we would type
% echo $PATH   (to see path)
% export PATH=$PATH:/newpath
- or -
% export PATH=/newpath:$PATH
- the two are different!
- first puts the new path at the end
- the second puts the new path at the front
- is important because when searching for the cmd to run, the shell will take
the first one that matches, so you want to make sure its matching the right cmd
- unfortunately you would have to do this every-time you logged in
- would get boring
- maybe there is an easy way fix this problem?
- #ask crowd for suggestion
- real answer: this is why you have dot-files
- we know dot-files hold configuration information
- zsh is no exception to this rule
- zsh has several files that hold login information
  - .zprofile
  - .zlogin
  - .zshrc
  - and more
- #depending on default setup will want to show how to do this add path
- the one that you care about is .zshrc
- once you change your .zshrc it won’t immediately take effect
- either you have to log out and restart your shell or you can apply the change from the cmd line
- to apply from the cmd line use the source cmd
  % source .file
- will cause the shell to read the configuration information, and apply the changes
- but this doesn’t cause the shell to reset its old configuration, it just overwrites what it changes
- so your PATH will be two copies of your old path followed by the added directory assuming you put it at the end
- not always what you want, and shouldn’t cause any problems unless you’re doing some funky things in your .zshrc or .zlogin
- *changing the prompt*
- well if you can change your PATH what other things can you change?
- lots of things, but the most fun to change is the prompt
- #demonstrate some nifty prompts in zsh and how to change them
- *other shells options*
- shells are complex beasts and offer loads of options, one of the most annoying is the shell beep
- to alert you to an error when you mistype something, the shell will cause the computer to emit a beep
- this is the most annoying thing ever
- to turn it off we would use the setopt command that zsh has
  setopt nobeep
- we would put this line in our zshrc; by default it should already be there
- there are lots of options so check them out!
- #may want to read through the default zshrc to see what’s set

• Using alias in the shell
  - it is a builtin cmd that allows you to “alias” one set of commands to another
  - so you can refer to a complicated, or repetitive, command with a smaller, easier to type alias
  - you always want to use these when you are typing some long string over and over again that you cannot just tab complete
  - is very much like a macro
  - what is one common thing that you might be typing?
  - how about “ssh username@turing.cs.hmc.edu”
  - so to create an alias we need a name: how about sshtoturing
  - from the cmd line would type: (spacing is important)
  % alias sshtoturing='ssh username@turing.cs.hmc.ed'
this alias is, of course, only good for so long as the shell is open
once you quit the alias is gone
the basic syntax is

```
% alias name='cmd arg1 arg2'
```

aliases also can’t refer to environment variables, have to do something
different if you need to use those: functions (not what we’ll talk about)
but what would be a good name? something better than sshtoturing

the whole point of the alias (for users) is to make your life easier
so want to choose a simple name that is easy to remember

how about: turing

says what you want, is simple and short

what would have happened though if we’d aliased this to ssh?

question of precedence...and the alias wins

in zsh and bash when you type something on the cmd line that you want to be
taken as a cmd, then if that text string has been aliased to something else
it will be replaced

is this a good thing or bad thing?

used in the appropriate places is OK

one example is ls

by default what you get on turing is not /usr/local/bin/ls
instead you get “/usr/local/bin/ls -F”

#do an example of what each outputs

#discuss why this might be a bad thing, tell about -G on GNU ls (hide groups)

#should watch which options you pass by default

so how did I know that all of you had this?

demonstrate the defaults alias’ that people have in zshrc
- for certain programs you’ll want the same behavior every-time you login
- so to make an alias more or less permanent you would put it in your zshrc
- so if you wanted the alias we talked about before you would add this line
  alias turing='ssh mkegel@turing.cs.hmc.edu'
- spacing is important so follow closely

• More about zshrc
- so the zshrc can set options and aliases for you, what else can it do at login?
- the at login is important since you can do this all from the cmd line;
  just gets old if you want it every time
- another random topic - creating files
- we’ve talked about permissions before but what permissions does a file get
  when it is created?
- by default every permission is ON, so it’s your job to turn them OFF
- the umask is the thing that does this
  - #show in zshrc
  - written out in octal permissions (remember those?)
  - #write out the permissions of the umask
  - #ask what the permissions of a newly created file would look like
- #ask if the results are good or not
- #what might be a good umask; what would be a bad one?

• UNIX programs for Windows geeks
- there are many different tools that unix offers
- you should now know how to make them work together and some of why they work
  as they do, but don’t exactly know what tools are out there
- *the programs you need*

  - #these are almost all graphical tools and are essentially replacements for programs that students would be using on either their Mac or Win box

  - web browser - firefox and mozilla, konqueror
  - text editing - jedit, emacs, vi
  - text processing - abiword, open office
  - document creation - latex
  - spreadsheet - gnumeric, open office
  - music - xmms, amarok (my favorite)
  - movies - totem, xine, mplayer, vlc
  - graphical terminals - eterm, aterm, xterm, rxvt, gnome-terminal, konsole
  - mail - thunderbird, evolution, pine/mutt through xterm
  - pdf viewers - xpdf, kpdf, gpdf
  - ftp - ncftp or lftp (cmd line), gftp
  - cd burning - k3b
  - aim - gaim, kopete
  - bit torrent - azureus
  - file manager - konqueror, nautilus
  - spell checkers - ispell and aspell
  - Windows like desktop - GNOME, KDE
  - window managers - fluxbox, xfce, window maker, enlightenment

- Three programs you’ll need to run your own system - su, sudo, and mount

  - *sudo and su*

  - #discussed before how use

  - su allows you to assume the role of the superuser (aka root) without having
to directly log in as root
- su <username>
- allows you to become that user, but have to know their password
- if you have first become root, you can do su <username> without a password
- sudo allows you superuser powers without having to become root
- sudo is flexible, you can have limited powers, or have the whole shebang
- sudo privileges are controlled in /etc/sudoers
- is a regular text file, like almost all configuration files on unix
- can add powers for users, or groups
- generally only modifiable by root (kind of a security risk otherwise)
- #demonstrate adding

mkegel ALL=(ALL) ALL
%wheel ALL=(ALL) ALL
%special ALL=(root) NOPASSWD:/usr/bin/myprog

joe_user local_host=/sbin/prog
- the basic format is: <user/%group name> <host>=[(who to run as)] <cmds>
- #see page 992 of unix power tools
- why would you ever want/need to use sudo?
- *mounting partitions*
- when managing a system you will almost always be modifying “files” that “normal” users should never touch
- these files are generally configuration files
- one of the most basic things to configure on a newly installed linux system are the partitions and how the hard disk will be managed
- in windows separate partitions would just magically show up
- more too it than this, but the management of the disks was done mostly
by the operating system

- also in windows the most the you would really ever do is put the disk/partition at G:, H:, or so on
- other devices were located at D: (generally a cdrom or dvd drive)
- A: - floppy drive
- C: - your main disk - the disk you booted off of
- these “locations” are known as mount points
- in unix you don’t have a:, c:
- instead disks and other devices can be “mounted” most anywhere
- that is if you have a newly installed hard drive you have to decide where in the filesystem tree that drive will be accessed
- this is what makes the file system(s) transparent to both users and programs
- #draw out a diagram of how multiple drives/cdroms/devices can be accessed
- generally mount stuff in /mnt, but can be anywhere
- can also mount what you might consider important directories from other drives
  - for example: /home on my system is another partition that I created for the express purpose of holding everything that I would want in my home directory
  - / holds everything else I need for my system, but is separate from what I have in my home directory
- this is a nice because it is very flexible and completely transparent to you and to any program that you might run
- one example of this being done is with each of your, and mine, home directories on turing
- your home directories are mounted locally on turing when you log in, but where they were before that doesn’t really matter
- obviously turing can’t have enough physical space to hold everything that we might want - home directories, staff accounts, projects, random data, backups
- all this disk space is provided transparently across a network through something called NFS or network file system
- so your home directories are located on one machine, while you are working on turing and doing things to your files, editing, compiling, etc.
- however that was just a setup for what you’ll likely do on your own machine
- *how to mount filesystems*
- in linux, we won’t worry about bsd or unix, devices are located in a /dev like we talked about before
- a device is of course just a file
- one thing to note: the linux 2.4 kernel is different from the 2.6 kernel in one major way – the 2.6 kernel does not use the /dev pseudo-filesystem approach to devices
  - in the 2.4 kernel every device that you could possibly have attached to the system was enumerated and given a mount point in /dev
  - in 2.6 the kernel is a little bit smarter about devices now and can handle them with some intelligence
  - for you this won’t matter too much unless you’re trying to hook in some exotic device
  - or if you have sound/video/usb problems (or want to use these devices) then you’ll get to know /dev pretty well
- generally a device will be owned by root and in one of several groups: disk, video, audio, root, etc.
- hard drives are generally given the designation hdXN, where X is [a-f] and is the drive and N is [0-9]
- so hda1 would be the first partition on the master hard drive
- hda2 would be the second partition
- hdb would be the second hard drive
- generally if you just say hda, or hdb you mean the entire device and not any one partition
- scsi drives (like what you’d have under firewire) are denoted the same as hard drives except with sd instead of hd notation
- another would be /dev/input/mouse0 (on my system at least) for the mouse
- most often though will deal with hard drives and partitions or other comparable (NFS) things
- so then how do you get a partition mounted?
- easy with the mount cmd
- mount is fairly smart, can usually figure out which file system is on the device
- note to windows users: Most any computer with XP will use NTFS which is a filesystem that linux can only read, and has trouble writing to
  - FAT32 will work just fine
- to use mount you need a directory on which to mount, so if we were to mount /dev/hda4 on /mnt/windows, then windows would need to be a directory under /mnt
- also to use mount you have to be root or have root powers (sudo)!
% mount /dev/device /mnt/mntpoint
- to specify the file system use the -t switch like such
% mount -t vfat /dev/hda4 /mnt/windows
- to un-mount the device use the umount command like such
% umount /dev/windows
- * working with partitions *
- when you are setting up a system its quite helpful to know which tools are available

- most systems you get will either come with windows installed, in which case the entire hard drive is one windows partition

- or you got a new hard drive

- (Windows Case)
  - You’ll want to get your hands on a program called Partition Magic
  - if you know where to look on the network (ffs.muddstudents.com *ahem*) you might be able to find the program
  - it can do most anything that you’ll want, including converting, moving and resizing Windows partitions
  - you probably shouldn’t let it touch any linux partitions

- (New HD Case)
  - you’ll want to use cfdisk or fdisk to create new partitions
  - use mkfs to then format the partition

- (General Case)
  - the cmd df will show how much space is available on each mounted filesystem
  - the cmd du will show you how much space something is taking up
  - useful alias

    % alias sizeof='du -sh'

- * Mounting devices with less pain *

- So I mentioned that I have /home mounted from another partition

- this happens on boot, but how?

- /etc/fstab is a configuration file that lets the system know which devices are to be mounted, where, who can mount, and with what other options

- #show my /etc/fstab - point out what each column is
- there are other things in things in fstab than just hard drives, but you can ignore them
- to modify fstab you have to be root (wouldn’t want random users messing with this)
- fstab tells the system which filesystems are to be mounted at boot time
- also lets the system know which mount points the system has
- so say I wanted to mount windows then all i would have to type would be
  % mount /mnt/windows
  - and the system would auto-magically know what to do
- you also use fstab to mount NFS exports, but we won’t cover that here
• Using cron

- cron is a daemon that runs in the background and schedules tasks
- such tasks might be backing up a hard drive, updating the locate database
- this configuration information is stored in a crontab file
- generally you just give crontab a file containing what you want done
  and when - this can be done as a regular user

% crontab mysched
- also, edit the crontab with crontab -e
- mysched will then contain single line entries for each task that you
  want cron to do

- each line is formatted like this

  mins hrs day-of-month month weekday [username] cmd

- the username is only there in BSD systems, but check your local docs first
- you can give a single number (7), range (3-8), list (5,7,9)
  or * to indicate all valid numbers for a given entry

- mins: 0 to 59; hrs: 0 to 23 (military time); day: 0 to 31; month: 1 to 12
  weekday: 0 (Sunday) to 6 (on BSD it goes from 1-7 with 1=Sunday)

- #show a random cron example

- cron can also make your life easier by just running things on a consistent
  schedule - say you need to run things on a hourly, daily, weekly,
  or monthly basis

- cron will run scripts that you put into /etc/cron.xxxx/

- where xxxx is daily, hourly, etc

- on those intervals

- you’ll have to be root to put stuff in these directories though
• Wine - Windows “emulation” for linux

  - *how to run windows programs under linux*

  - use wine!!
  
  - there isn’t much else to tell you
  
  - install wine
  
  - do this from the cmd line

  % wine WinApp.exe

  - hope for the best; does not work for every application

  - if you want to run games (or itunes, or quicktime)

  then you’ll need a thing called cedega

  - there is good documentation, you just need to find it

• Sharing files under linux

  - use a package of programs called samba

  - allows you to share and access windows network shares

  - really easy to use but has issues with some of the newer versions of

    the windows sharing protocol - may not be able to access all files

• Emacs and vi

  - please use emacs, or if you have a graphical environment something like jedit

  - vi should only ever be used in the most desperate of circumstances

  - emacs and vi

    - #hand out reference cards

    - #demonstrate some nifty tricks (M-q for one)

    - #how to exit and save
- #F10 is menu in emacs
- #vi is a modal editor - some people “just get this” way of editing
- not our goal to teach students to use these

• More topics to talk about
  - shell scripting
  - perl and python
    - both powerful interpreted languages
    - easy to learn and use; generally only need to know one but should learn at least one
    - checkout these sites:
      - diveintopython.org and python.org
      - perl.org and perl.com
  - one of the best things to learn are unix regular expressions
  - #demonstrate this and have examples and exercises
  - two powerful tools are find and grep
  - #will want demonstrations for both
  - two more tools are sed and tr
  - #demonstrations for these too
  - screen is a helpful tool for managing terminal screens
  - logging in through ssh
  - how to X forward with ssh (-X | -Y)
  - how to use ftp/sftp and scp
  - why ftp is a bad idea - any password send in the clear