Introduction to UNIX

Logging in
Basic system architecture
Getting help
Intro to shell (tcsh)
Basic UNIX File Maintenance
Intro to emacs
I/O Redirection
Shell scripts

Logging in

- most systems have graphical login
- on Linux machines
  - log in to local machine
  - for now Gnome best choice for window system
- Department has lots of SunRay terminals
  - not standalone workstation, connect to central server

- system asks for username and password
  - password is only "real" thing stopping others from breaking in to your account
  - change password periodically, passwd(1) command
  - need to change it on sol.cse.buffalo.edu
  - make sure not a dictionary word

- windows and applications may or may not appear automatically at login
  - remembers some state from previous login session
  - sys-admins may have some things run automatically
  - can customize Gnome quite a bit

- STRONGLY encourage using SSH-based "client" for remote access
  - network traffic encrypted
  - handles "X11 tunneling" and display access issues automatically
  - login from UNIX to UNIX using slogin(1) or ssh(1)
  - MacOS-X comes with SSH server/clients (MacOS-X core is UNIX)
  - SSH clients for Windows available (CIT)

Basic System Architecture

- core of UNIX known as "kernel"
  - handles hardware
  - creates small number of "initial processes", manages processes (timesharing)
  - provides processes controlled access to "system resources" (files, timers, sockets, etc) through system calls
  - has from UNIX's beginnings been designed to be multi-user multi-tasking system

- utility programs provided to do various tasks
  - to some extent vary from one UNIX to another
  - mv(1), cp(1), rm(1) to manipulate files
  - mkdir(1), rmdir(1) to manipulate directories
  - csh(1), tcsh(1) for command shell
  - vi(1), emacs(1), pico(1) for editing files
  - many other utilities to do many other things
  - (1) after words above indicates section of Manual, will talk about UNIX Manual later

- UNIX windowing systems based on X11
  - currently only major exception MacOS-X which is based on “Aqua” but X11 for MacOS-X available
  - X11 packages available for Windows (Xwin-32 available from CIT)
- two main components to X11 systems
- “Server” runs on local machine
  - controls display, keyboard, mouse
  - provides some sort of (hopefully authenticated) access

- “Clients” connect to server
  - use X11 protocols to tell server how to display things on screen, read from keyboard, get mouse events, etc
  - X11 protocols completely network based, clients can run on local machine or a remote machine
  - may need to authenticate to server as part of initial connection

- variety of “Windowing systems” built on top of X11
  - how server and initial set of clients get started determines windowing system in use
  - will focus on Gnome for now
  - others popular on various UNIX's include KDE, Gnome

Getting Help

- if all else fails (or you get sick of trying) send email to cse-consult@buffalo.edu
- the Department’s “Service Catalog”:
  - https://wiki.cse.buffalo.edu/services/

- UNIX Online Manual pages
- most useful if you already know name of command or other UNIX system resource
  - can use keyword search but output is often lengthy (e.g. use “man -k owner | more”)
  - “man intro” includes among other useful basic information a list of many UNIX utility programs

- Sections available vary slightly among Unixs, set for Linux are:
  - 1: commands
  - 2: system calls
  - 3: library functions
  - 4: special files (mostly devices)
  - 5: file formats
  - 6: games/demos
  - 7: misc. stuff (didn't fit in any other section)
  - 8: system administration commands

Intro to shell (tcsh)

- shell is command interpreter
- first thing on line command or alias
- commands typically have "options" and "arguments"
  - options have a "-" in front of them, usually effect the way the command runs in various ways
  - arguments are things needed by command to do its job (e.g. names of files you want rm(1) to remove)

- many commands take "--" to mean stop processing options and treat rest of command line as arguments
  - e.g. needed to remove file named "-t" because rm(1) tries to treat "-t" as command option instead of argument and gives a usage error message
- all options and arguments for any given command described in (surprise!) command's Online Manual Page

- can use "wildcards" when trying to specify filenames
  - * : matches any set of characters, use with caution! (command "rm *" removes everything)
  - ? : matches any single character
  - [a-z] : matches any one character in range a-z (ASCII character codes)
  - wildcarding will ignore filenames beginning with "." unless you force it, e.g. "rm .backup*"

- "tab completion" can save LOTS of typing
- for filenames can type part of filename, then press tab key
  - if what you typed so far uniquely identifies one file shell will complete entire name for you
  - if multiple names match what you typed so far shell will complete as much as it can and beep to indicate completion is not complete
- can type control-D instead of tab, shell will show you on screen what name(s) match so far

- can use tab/control-D for commands as well, shell will look through your path and aliases for possible completions instead of treating it as a filename
- up-arrow key on keyboard OR control-P shows you previous commands
  - can edit command line that results using back-arrow, backspace, typing in new pieces, etc.
  - pressing carriage return executes resulting command line

- shell allows aliases
  - "permanently" alter how existing commands work

    ```
    alias ls 'ls -AF'
    ```
  - rename commands to your liking

    ```
    alias dir 'ls'
    ```
  - create new command from existing ones

    ```
    alias rmobj 'find . -name \*.o -exec rm {} \;'
    ```
  - be careful to not name alias something that is important (e.g. weird things happen if you make alias named "set")

- shell stores list of directories it will look for commands in (path) as a shell variable
  - shell variables identified by preceeding name with "$", set using set(1) command
  - shell has some special variable names it pays attention to, e.g. prompt is "$prompt"
  - list of currently set variables can be seen with set(1) command, no arguments
  - can use echo command to see just one variable, e.g.

    ```
    echo $path
    ```

- should minimally include /usr/bin, almost all normal UNIX commands that are part of baseline distribution go there
- sys-admin's typically add many more directories
- to add in your own ALWAYS just add to what is already in path:

  ```
  set path = ($path ~/bin)
  ```

- note that path is just a "convenience", can always type full pathname to command :

  ```
  /usr/bin/vi .cshrc
  ```
- which(1) command shows where shell runs command from, some UNIX utilities have multiple (different) versions

- shell builds list of commands from path when it first starts, if commands get installed after you log in may need to use rehash(1) to have shell rebuild list of commands
- path is special, shell makes it appear as both $path shell variable and $PATH environment variable
  - shell variables normally only visible to currently running shell
  - environment variables get passed on to all child processes (commands shell runs)

- use setenv(1) command to set environment variables, except for PATH environment variables totally different from shell variables
- setenv(1) with no arguments shows all currently set environment variables
- commands may look for certain environment variables and alter behavior based on setting
  - e.g. some commands start editor for you, can use environment variable EDITOR to specify which editor

- commands that look for environment variables will list which one(s) and what they effect in ENVIRONMENT section of manual page
- most common ones EDITOR and VISUAL to specify editor, e.g.

```
setenv EDITOR xemacs
setenv VISUAL xemacs
```

- when tcsh(1) starts it reads file named ".tcshrc" in home directory or, if ".tcshrc" doesn't exist, ".cshrc" for backwards compatibility with csh(1)
  - can add set(1), setenv(1), and alias(1) commands to .tcshrc file so they happen every time you log in

Basic UNIX File Management

- cp(1) copies file(s)
  - source file first argument, destination file second argument
  - if last argument directory copies file(s) into directory retaining name
- mv(1) moves file(s)
- rm(1) removes files
  - by default won't remove directories
  - command flag "-r" will recursively remove directory

- mkdir(1) creates directory
- rmdir(1) removes (empty) directory
- cd(1) changes current working directory
  - pwd(1) shows current working directory
  - directory "." is always current working directory
  - directory ".." is always parent directory
  - "~" expands to your home directory if followed by "/", e.g. "~/bin"
  - *~username* expands to home directory of *username*

- ls(1) shows directory listing
  - by default just file and directory names
  - does not list names beginning with "." (use "-a" flag to see those)
- longer listing shown by "-l" flag, includes
  - file type, permission settings
  - link count
  - owner/group
  - size, date last modified

- read, write, and execute permissions can be granted to
  - file owner
  - users in group file is in
  - everyone else
- chmod(1) used to change permissions

Basic emacs

- two ways emacs uses to separate characters meant to be in file from characters meant to be commands
  - hold down control key while pressing character, e.g. C-u means hold down control key while pressing "u"
  - preceed character by pressing "Escape" key, e.g. M-v means press escape key, then "v" (M- is called "Meta")
- some keyboards have Meta key

- first thing to learn is how to start emacs
  - most people prefer xemacs version of emacs, just type "xemacs" as command
  - if working through non-graphical connection use "xemacs -nw" for "No Windows"
- second thing to learn is how to exit emacs
  - C-x C-c will exit emacs, can save or not save changes
- most commands can be repeated *number* of times with "C-u *number*" before command

- typing characters without C- or M- enters them into file
- several ways to move cursor small amounts
  - arrow keys on keyboard usually work
  - C-n moves down one line
  - C-p moves up one line
  - C-f moves forward one character (right)
  - C-b moves back one character (left)

- can also move in larger amounts
  - C-a moves to beginning of line
  - C-e moves to end of line
  - C-v moves forward one screen full
  - M-v moves backwards one screen full
  - M-> moves to bottom of file
  - M-< moves to top of file

- in graphical mode
  - left-clicking over text moves cursor there
  - scrollbar works
  - left-click, hold button, drag mouse, release button will highlight area (used for more advanced editing later)
  - right-click brings up small menu of operations

- C-x u will undo most recent change
  - repeating will keep undoing changes as they had been made, often can undo every change made to the file since start of editing session
- C-g aborts almost anything
- mark/point/region used more advanced features
- point is wherever cursor is right now
- mark can be set "manually" or as a side-effect of some commands

- C-space (hold control key, press space bar) sets mark to be where cursor is now, remains there when you move cursor
- region is piece of file between mark and point
- in graphical mode left-click-hold-button-drag-mouse-release-button defines region
  - where button originally pressed becomes mark
  - where button released becomes point

- insert text by just typing characters
- couple methods to remove text:
  - C-d removes character cursor is on
  - Backspace removes character to left of cursor
  - C-k removes from cursor to end of line
  - C-w removes all text in region
    - text removed this way gets placed in “kill buffer”, C-y will insert text from kill buffer at current cursor position
    - used to move or copy blocks of text

- can be editing (or viewing) more than one thing at a time in different “buffers”
  - xemacs starts showing one window with file you gave on command line, or scratch buffer if no filename
- C-x C-b will split screen, shows buffer list in bottom half
- cursor will only be in one window at a time, that is “active” window
- C-x o moves cursor to other window

- C-x 1 removes other window, active window takes up full screen
- C-x 0 removes active window, other window takes up full screen

- all things emacs can do have named functions
  - command keys “bound” to named functions
  - e.g. C-n bound to function “next-line”
  - M-x will drop cursor to “mini-buffer” at bottom of screen, can type in any emacs function name
  - can type part of command and use tab key to “investigate” possible completions

- can have different modes depending on what is being edited, personal preferences, etc.
- may start off in special mode based on filename (suffix)
- can use M-x to adjust modes
  - e.g. M-x auto-fill-mode turns on automatic line wrapping
  - “fundmental-mode” has no special features

- C-h enters into help system
  - need to enter another character for help mode
  - new users can go through online tutorial by entering "C-h t" (HIGHLY recommended!)
  - C-h followed by two "?" will bring up complete list of help modes
- Info system also has lots of information
  - C-h i enters Info mode (or use Info button in graphical mode)

I/O Redirection

- three "file descriptors" most UNIX commands use
  - read from "standard input"
  - write normal output to "standard output"
  - write error messages to "standard error"
- most often all three default to login terminal
- can have shell change defaults as part of command line

- to make file standard input to command with "<" character:

```
% mailx kensmith < message
```

- to make file standard output of file use ">" (overwrite) or ">>" (append)

```
% cat file1 file2 file3 file4 > bigfile
```

- to capture both normal output and error messages use ">&":

```
% make >& MAKE_OUT
```

- can use pipe to connect standard output of command on left to standard input of command on right:

```
% grep deact /etc/passwd | wc -l
```

- can use pipes to piece together simple UNIX utilities to do complex tasks

Shell Scripts

- UNIX has two forms of executable files
  - directly executable, contain "machine code"
  - shell scripts, contain text to be interpreted
- directly executable files built by linker, last phase of compiling program source code
- shell scripts edited with normal text editor
- first line of shell script identifies what program should be used as the interpreter

- when UNIX starts up shell script it:
  - reads first line to get pathname of interpreter
  - loads interpreter – interpreter MUST be directly executable file (machine code)
  - runs interpreter with standard input of interpreter being the shell script file
- shell script file should contain "language" appropriate to the interpreter

- standard “hello world” program as a C-shell script:

```
#!/bin/csh -f
set path = (/usr/bin)
echo “Hello World.”
```

- most UNIX shells offer all the features that make up a programming language
  - variables
  - conditionals
  - looping

```
#!/bin/csh -f
set path = (/usr/bin)
foreach i ($*)
  grep ^${i}: /etc/passwd > /dev/null
  if ($status) then
    echo “${i}: No such user”
  else
    grep ^${i}: /etc/passwd | awk -F: '{print $5}'
  endif
end
```

- most popular scripting language these days is perl, check into that if developing new script programs
- need several lectures' worth of another course to cover shell scripts in any more detail than this :-)