# CS 35L Lecture Notes

# Lecture 1 – wk0 9/23/21

### General Information

**General patterns of how to build software:**

This includes scripting, dependencies, version control, self reference, debugging, and client-server. These will be the major case studies of this course.

**Grading:**

* 24% final midterm – open books + notes (no computer so print out notes)
* 15% midterm – open books + notes (no computer so print out notes)
* .5% two feedback on Las
* 1.5% class participation
* 35% main project
* 24% homeworks

**Linux:**

Use lnxsrv11.seas.ucla.edu, lnxsrv12.seas.ucla.edu, lnxsrv13..seas.ucla.edu, lnxsrv15.seas.ucla.edu for linux.

Set path environment using PATH=/usr/local/cs/ban:$PATH

**General Principles of this course:**

This course will stress principles of programming, data structure design, integration and gluing together of code, configuration, testing, version control, security, and forensics.

### Emacs, POSIX shell, POSIX File System Organization

These systems address problems and issues related to

* Functionality
* Reliability (when power goes off, application crashes, etc)
* Performance – performance and reliability is a tradeoff, neither can be fully satisfied
* Understandability – can we look at how code is implemented and understand why it was done that way without running into significant roadblocks

# Lecture 2 – wk1 9/28/21

### POSIX

POSIX file system (Bell Lab 1970s + UC Berkeley ~ 1970s)

POSIX shell (S. Bourne Bell Labs 1970s)

Emacs (MIT 1970)

POSIX --- Portable Operation System IX – specification series 1980s.. IEEE etc, 2018.

**The shell can operate as:**

1. **Read Eval Print Loop (REPL)**
   1. A command can be read, evaluated by the press of enter / return, printed the result of the evaluation, and then loop back and ask to read again.
2. **Script**
   1. A script will usually be found in a file, and it is somewhat different than REPL.

### Emacs

Files live in the file system and are persistent, which means that they will not be deleted till explicitly done.

Emacs have buffers that live within emac’s RAM, which means your work will be lost if you exit out of emacs. This is done due to:

1. Performance – you only change and work with buffers, which is significantly faster
2. Functionality – keep track of what you are working on distinctly from a file or commit somewhere

**Emacs cheat sheet:**

(meta(aka alt)) M-x shell – start emacs terminal

C-g – escape / interrupt / quit

C-x o – switch to other buffer

C-x C-b – list buffer

C-x k RET – killing off (deleting) the buffer

C-x 0 – stop displaying the buffer, but still exists

C-z – OS process suspends emacs

C-x C-c – exits emacs

C-x b NAME RET – switches to buffer NAME, creates one if it doesn’t already exist

ls – list contents that can be visible

ls -altr – means list all

C-x 4b “ ‘ ‘ -- in another “window”

C-x C-s – save

C-x 2

C-x 3

C-h

C-h b – list bindings of keys

C-h l (letter L) – view lossage (recent commands)

C-h k C-x 3 – gives you any and every command

C-h t – tutorial

TAB – completion

C-@ -- mark current location

M-w – copy all between mark + point (cursor location

C-y – yank (the stuff that was saved)

M-q – indent

cd NAME – change working directory to directory NAME

cd . -- goes to the children

cd .. -- goes to the parent of that file (root is its own parent)

cd ../../… -- goes to the gradparent

ln A B -- creates a name B for the existing file named A

rm A – removes the name A from the ls directory

mu A B – removes directory entry A to B

ls -ail -- lists all files / directories

ln -s – creates a symbolic link from a file using a string as the symbolic link

A buffer is an array of text and meta data.

Jobs command is your shell’s subprocesses, and ps is for your processes or anybody’s processes.

Most files do not correspond to buffers, and when in emacs, a lot of the buffers won’t even correspond to files.

A directory is a partial function from file name components to files. These files are contained in a tree-like structure.

# Lecture 3 – wk1 9/30/21

The file system is built off indirection – inside file system there is a unique integer associated with every file in the system.

A directory maps each name to a file number which creates the tree-like structure of file systems. And by definition, directories are files too because it too contains integer values to be mapped to one another directories.

“Any problem in computer science can be solved with another level of indirection” – obviously not entirely true, but indirection is a powerful device that can make an illusion of data structures but underneath is just a bunch of tables.

There can be no hard links to directory except for ‘.’ and ‘..’, or links from parent.

Symbolic links can indirectly link directories and files to other directories and files.

There can only be a max of 20 symbolic link chain

A regular file contains a finite sequence of bytes, and in the posix model there is a sequence of 8 bit bytes. This file also has meta information that is associated with a file which contains timestamps (last accessed, read/write, creation time, etc), ownership information, group information, link count, and permissions (12-bit integer that encodes all that you are allowed and not allowed to do with the file).

C-h k C-x C-f

A large character set using byte strings is called a Unicode. UTF-8 is an example of the usage of a Unicode.

Emacs terminology –

* File (like POSIX) – persistant sequence of bytes sitting in file system
* Buffer – string of bytes sitting in emacs RAM
* Window – view of a buffer
* Frame – a window (C-x 5 2: create frame, C-x 5 0: delete frame)

C-x d NAME Ret -- edit a direct to NAME

C-/ is undo

When you are editing a file:

* #F symbolic “to” the user editing the file
* F~ backup file (copy before I started editing)

Cd .snapshot

# Lecture 4 – wk2 10/5/21

### Modes of Interaction

**User interface tension:** Two goals of building a user influence ---

1. The goal is to make a simple, regular, always-the-same-thing interface.
   1. Want easy to learn, and easy to remember
2. Efficient user interface tailored for your task
   1. If you are an expert, the use is faster

One is going to have to design a product that attempts to meet 1 or 2 of these goals, if even possible.

A mode in emacs is an environment which you can set up for yourself or someone else does it for you, and this environment is tailored for particular tests. The downside is that things you used to type to do something, now does something else. Emacs is modeful—can be treated to tailor for your task.

To switch to another buffer, type C-x-b.

M-1 takes a standard buffer from part of a buffer in emacs and places into a file.

Per-buffer:

* M: -- runs a lisp code and print out value
* Point: There is a point that holds the cursor location M:point
* Mark: the place you’re marked C-@

Scratch:

* LF C-j evaluate-previous sexpr ‘\n’
* C-x C-e “ “ ‘ ‘ (only buffer)

M-w – copy

M-y -- paste

### Compilation & Integration

C-j takes the source code to the lisp program, creates an data structure representing the program. It will usually be of the form of a tree—and then the program traverses the tree. This is very slow due to number of pointers involved.

There is a byte code interpreter to resolve this issue, it stores the program in lines of byte code to do the operations at a minuscule and faster time. M-x byte compile distel RET 🡪 creates dist.elc which is machine independent.

$elc foo.el creates a compiler into machine code foo.o which can link into emacs. This is machine dependent however, but the upside is greater performance.

Compile the machine code at runtime, called just-in-time (JIT) compiling. Once JIT has been compiled, it will run at machine runtime speed.

C-h f what-cursor-position RET

C-x =

M-x what-cursor-position RET

M-: (what-cursor-position)

Global-set-key “ ” ‘ ’

# Lecture 5 – wk2 10/7/21

### Layers

#### There are multiple levels of Emacs –

* Emacs “itself”
* ELPA (larger set)
* Emacs list
* Emacs executable (.o, .c, .sc files) (C/C++)
* Syswall – Linux Kernal (between upper and lower level)
* Machine x86-64

These are stratified to create a level of abstraction to resolve your problems. You have to understand what level is required to solve your problem.

### Pattern Matching

**Posix Shell –**

There is a shell under the Unix operating system to give users control to anything they want under the operating system. This is the REPL c program (read, eval, print, loop) where user types something, shell reads, evaluates, and then prints the value and waits for the next command

Grep -- globally look for this regular expression and print

Write “grep foo/etc/passwd | grep” –

* Execute 2 execlp (calls grep)
* Fork() + pipe() system calls
  + All the real work is done by grep, which is from the OS, the shell just helps execute it

Aside on Grep –

* Ed (simple text editor)
* GNU grep is the most popular implementation – gives it a pattern, prints every line that matches. If you don’t list a file name, it reads from the film.
* A regular expression uses ordinary characters to stand from themselves.
  + a,b,c
  + RS concatenation of matches of R and S
  + R\* zero or more occurrences of R
  + . any char bot newline
  + [areiou]
    - Any char set
    - [a-zA-z0-9\_]
    - [a\*/-] – set minus to last otherwise it is seen as a range
    - [^a-zA-Z] – negate all chars in ASCII set
    - [[:alpha:]] – matches any alphabetic character
    - [[a] – matches [ and a
* grep \”\” /etc passwd
  + The \ makes it so that the shell doesn’t treat the single quote as a special char
* grep -E
  + new metasymbols
    - R|S “or” matches anything R or S matches
    - (R) just grouping operater
    - R+, RR\*

### Little languages

RE (regular expressions) are a shell language designed for pattern matching.

Sh is a little language that is designed for control of operating system.

Little languages and Unix / Linux / etc. [C Basis]

grep REs are a small language for pattern match.

* sh is a little language for control of OS.
* sed specifies a little language for editing data streams.
  + Sed ‘x/d’ deletes every line with an x
* awk specifies a slightly bigger language for editing data streams
* perl is the language that combines all the little languages into one

The shell is a tool for setting up an environment for a program to run in. This includes

* the working directory
  + cd (/etc && pwd) OR cd( /etc; pwd) – changes directory within the context of the command
* the std in, out, err, streams
* environment variables – set of name value pairs
* user id – “id” command lists your groups

shell builtin commands:

* . files
* cd dir
* exit 27

File names in POSIX:

* “/ “ start of root directory
* “[] “ anything that replaces [] start at this process’ working directory
* “” is an invalid filename
* A directory is a mind
  + If empty or “.” go past the /
  + If the file name component is “..” go to parent directory

# Lecture wk4 10.21.21

**Table Driven Programming –**

We can do things entirely in a program using ordinary if-then-else’s, but we don’t want to do that because it is often costly and unnecessary. Instead we can use tables:

Dictionary – table = {x: ‘abc’, y: ‘def’, z: ‘ghi’} 🡪 parse that information into inf = table.get(x, ‘null’) where the first element is the index and second index is the default

**SGML (standard generalized markup language) –**

This is a data language for publishers. This is not something you can write programs in so it is not procedural, rather it is declarative.

It specifies document structure, but not exactly how to format it. Some variation in the structure is allowed to allow the data structure be different for the respective data a publisher is attempting to display

Document Type Definition (DTD) – you can place elements into your document and the SGML doc will tell you the rules for those elements. There is a list of possible subnodes and a list of possible attributes and their values.

HTML = SGML + DTD for web essentially

HTML5 is a living standard, which means it doesn’t directly coincide with DTD’s that have been punished in the past

XML is extensible Meta Language – it is SGML without the publisher stuff, so it only focuses on how to represent trees of nodes and their attributes and values

DOM (Document Object Model) – easy way to access certain elements as it is like an API for investigating / walking through trees.

* There are entities and elements in these trees:
  + Entities are the “leaves” of the tree
  + Elements are the nodes of the tree
* You can modify the tree by adding/changing elements that is live
* Creating DOM objects on the fly with JS

**Brower Rendering Pipeline –**

No JS:

* Download entire document
* Parse into DOM
* Walk through it
* Decide where to place everything
* Paint the screen

Common optimizations:

* If the element looks like it won’t be visible to user, don’t render or execute the javascript code associated with it.
* If the browser says elt is low priority, don’t execute
* Guess layout of whole page for the top-level structure, and render based on that with re-renders as needed.

**CSS –**

CSS focuses on the presentation of a DOM, rather than provide content for the DOM. You can annotate parts of the DOM with style mods

**JSON (Java Script Object Notation) –** textual notation for object that are typically trees

**Node –**

Javascript based runtime for asynchronous events. Basic programming constructs is the event loop – node is waiting for an event to occur and that use that event to make some transfer of data.