정수민

Download source file

SLS Lecture 4 : Editors, Make and Terminal IDEs

Contents

- 4.1. RECAP
- 4.2. Computer Systems Conceptual
- 4.3. Computer Systems Conceptual
- 4.4. Running (Runtime) Structure of a Computer System
- 4.5. OS Kernel - UNIX Kernel as our example
- 4.6. OS Kernel - UNIX Kernel as our example
- 4.7. Computer Systems Practical – UNIX Example
- 4.8. Let's try putting it all together with an example
- 4.9. WHAT COMES NEXT?
- 4.10. ASCII Editors and Programming
- 4.11. Practical side of files
- 4.12. EDITORS
- 4.13. Make

and some shell script programming to boot

https://web.archive.org/web/20110407224426/http://jtnimoy.net/workviewer.php?q=178

4.1. RECAP

Let review what we have discovered so far

4.2. Computer Systems Conceptual

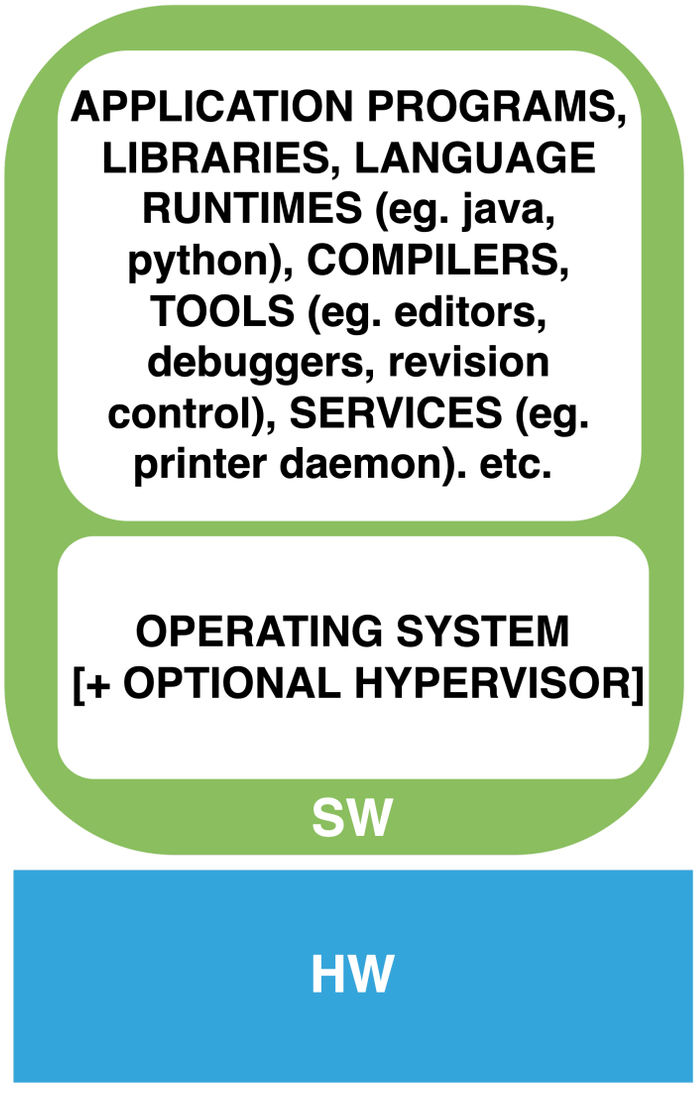
Breaks down in to two parts

1. Hardware
2. Software

4.3. Computer Systems Conceptual

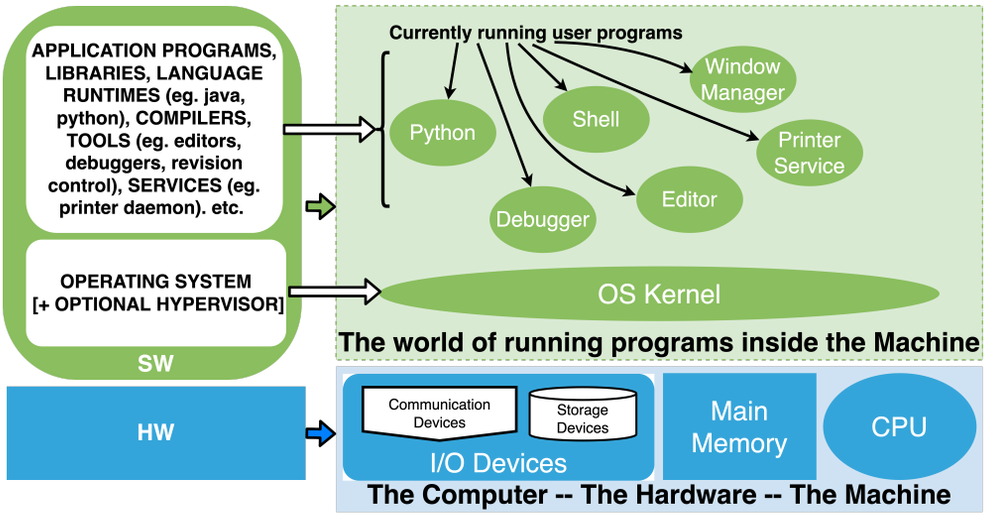
Software broken down

1. Operating System
2. User (Everything else)



4.4. Running (Runtime) Structure of a

Computer System

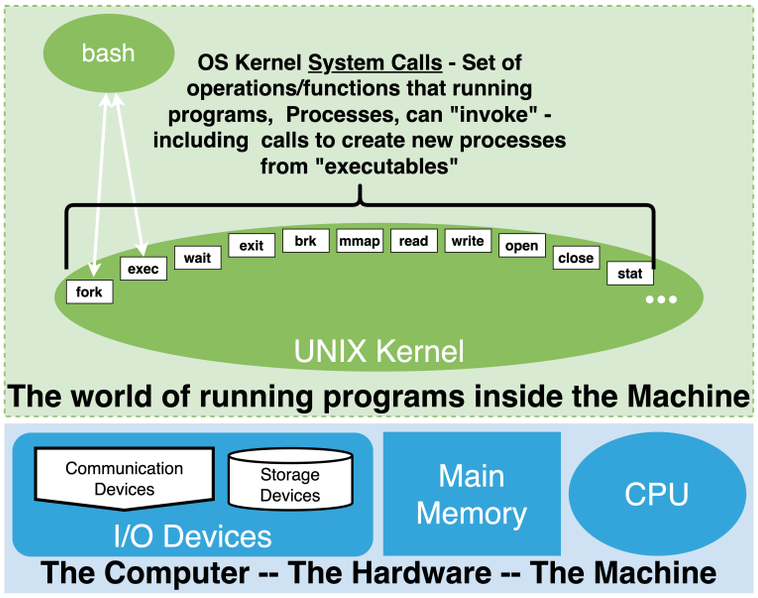


4.5. OS Kernel - UNIX Kernel as our example

what kernel do?
1. multiprocess the program

4.5.1. OS Kernel System Calls

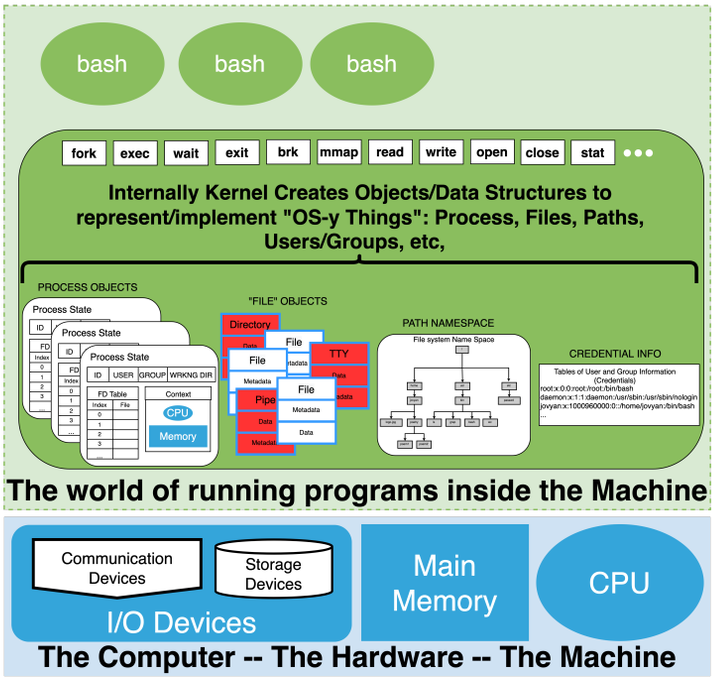
1. Processes talk to kernel through system calls
2. System Calls can take arguments and return values



4.6. OS Kernel - UNIX Kernel as our example

4.6.1. OS Kernel Data Structures and Objects

1. Kernel provides abstract objects like processes, files, tree of path names, users, etc
2. Internally, uses data structures to represent the necessary state



4.7. Computer Systems Practical – UNIX Example

1. ASCII Terminal provides a text based way for physically working with system
2. UNIX Shells, eg. bash, provides ASCII command line “language”
3. Core OS Ideas
    1. Processes:
        1. a “running” program
        2. shell external commands: launch processes
        3. signals: pause, un-pause, kill processes
        4. list: `ps`
    2. Paths:
        1. hierarchy of names : directories and files
        2. process have a working directory
        3. full path vs relative path
    3. I/O: Files: data and metadata
        - Types (typically found on all UNIXs)
            1. regular files : the “normal” files
            2. directories
            3. devices files: eg. terminals (tty files), ssd’s, etc
            4. pipes
            5. sockets
            6. symbolic links
        - Streams and redirection
            1. process open files as a stream
            2. stream identified by numeric index, fd
                - 0: standard input → key board
                - 1: standard output
                - 2: standard error
            3. shell redirection syntax
                - setup streams
    4. User and Group Identifiers:
        1. user id 0: Root user
    5. Permissions
        1. File Meta data indicate owner and permissions
        2. Processes have credentials

3. Accesses are permitted if process credentials and permissions match

---

4.8. Let's try putting it all together with an example

- piping cats

In a single terminal

```
cat | cat
```

and illustrate

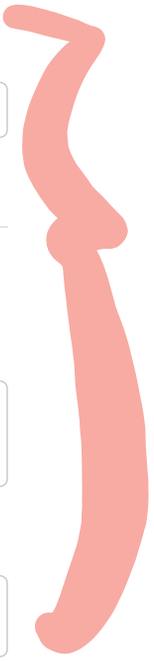
Connect cat in two different terminals

Terminal 1

```
mknod mypipe p
ls –l
cat > mypipe
```

Terminal 2

```
ls –l
cat < mypipe
```



4.9. WHAT COMES NEXT?

In order to learn to write "real" software we need some tools

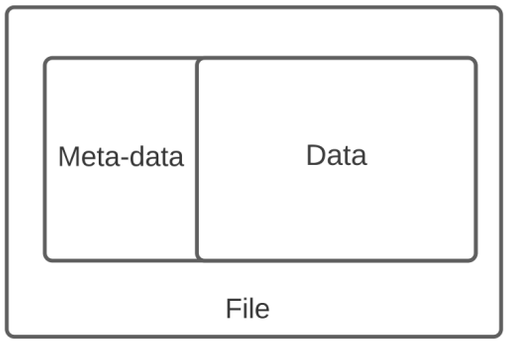
1. Shell organizing and navigating files
2. ASCII Text Editors - `emacs` (or `vim` if you must)
3. Build System - `make`
4. UNIX as an IDE - organizing your terminal windows
5. Re-vision control - `git`

Then we will be able to integrate debuggers, assemblers, linkers, and compilers

---

4.10. ASCII Editors and Programming

- We now know what a regular File is right?
- ASCII file: is simply a file who's data is encode in ASCII
- Traditionally HUMAN "friendly" program source code has been written as ASCII files - Why? so we can read it



A file is an abstract object that can store Data and has associated with it additional descriptive facts we call Meta-data

Types of Paths

```
b       block special
c       character special
d       directory
f       regular file
l       symbolic link
p       FIFO
s       socket
```

- We now know that what we think of as a file is a bit subtle and depends on context
  - generally, however, as a user we think “normal” files – as objects that we use to organize and store our information. Eg. File objects that are backed on devices which will hold our data until we explicitly delete it.
- ASCII again is a simple coding of the English language along with some control characters
  - code make it easy for us to “parse” the contents
    - where each token of the coding is mapped to a single byte value
- Given ASCII’s simple coding it is easy to write programs that can “process” its contents
  - thus we can specify a grammar/syntax/language using ASCII. Eg. python, shell script, C, java, etc.
  - that we can then write programs to parse and transform into other representations that can then be “executed”
- UNIX has many man tools for working with ASCII files

4.11. Practical side of files

- We should know how to use our OS to “manage” files
  - explore what files exist and navigate the directory tree
  - know how to use, full and relative, paths
  - create and remove directories
  - create files, redirect a commands output into a file, and read data from files
  - remove files
  - search ASCII files
- lets put all this knowledge to good use and write a program – shell script

- Navigate using `pwd`, `cd`, `ls`, using full path `ls $HOME/bin`
- `mkdir`, `rmdir`, `rm -r`, the dangerous `rm -rf`
- `touch`, `echo "contents of  /bin:" > myfile`, `ls /bin >> myfile”
- `rm myfile`, `rm *.o`,
- `grep`, `rgrep`, `find`
- bigger example :

```
# How can I navigate to home dir?
cd ~
# lets get a copy of the program source code from the internet (github)
git clone https://github.com/kornia/kornia.git
# use find to print all files
find ~/kornia
# what is the difference between * and "*"
echo *
echo "*"
# use "-name" argument to tell find to restrict/filter only name that match
<anything>.py
find ~/kornia -name "*.py"
# search all of them for functions in a file
# What is the python syntax for defining a function?
#   def followed by a space
# What can we use to search a file for lines that has this "pattern" in it?
#  grep
grep "def " ~/kornia/conftest.py
# need a while loop to process each file
#   what is read? "help read" -- very powerful idiom combine read  with while
find ~/kornia -name "*.py" | while read file; do grep "defs " $file ; done
# lets turn this into  a program -- shell script
mkdir ~/bin
echo '#!/bin/bash' > ~/bin/findfuncs
echo 'find ~/kornia -name "*.py" | while read file; do grep "def " $file ;
done' >> ~/bin/findfuncs
cat ~/bin/findfuncs
findfuncs
# What are we missing?
chmod +x ~/bin/findfuncs
findfuncs
# What are we missing?
export PATH=$PATH:~/bin
findfuncs
```

same variable

find ~/kornia/ -name "*.py" | while read $file ; do grep "def " $file ; done

4.12. EDITORS

- Our program has a lot to be desired of it
- Pretty painful to keep going this way
  - although there are “stream” “editors”, `sed`, `awk` and `perl`
- Terminal based ASCII Editors and programming
  - two main UNIX Religions: VI(M) and EMACS (https://en.wikipedia.org/wiki/Emacs#Church_of_Emacs)
- in reality there are many others Terminal based editors including: `nano`, `joe`, `man -k editors`
  - many editors allow for “multiple panes” and can even have terminal panes within the editor
    - “pane” divides the UI into sub-areas that can each display their own content

4.12.1. Beyond VI and Emacs

- there are more modern editors that only have a GUI
  - sublime, atom, vscode, kate, etc
  - The Jupyter Environment is actually an IDE with its own editor designed for web-browsers

4.12.2. BUT...

It really pays to know how to be functional in VIM and EMACS at least a little

- it means that you can be highly productive even with just “tty”/terminal access.
  - eg. you have `ssh` terminal access to a remote computer
- core component in constructing your own low-level IDE (with terminals and other tools)
- understanding how things actually work and fit together
- lowest common denominator is still the terminal

4.12.3. VI and VIM

- VI 1976 by Bill Joy: https://en.wikipedia.org/wiki/Vi
- VIM 1991 by Bram Moolenaar: https://en.wikipedia.org/wiki/Vim_(text_editor)
  - VIM has in some sense subsumed VI and on most unix systems when you issue vi you really get VIM
  - Has some support for extensions and advanced features
- Remember all these tools were written by programmers for programmers
  - its greatest power and its learning curve comes from its text based command interface
    - AKA it is not menu driven
    - two core modes of operation: Command mode and Insert mode
    - It actually has 12 modes (according to the wikipedia entry)
  - I only know the basics but use it very often when need to make quick changes to files
- built in help `:help` and lots of online resources and tutorials – including a `game`
- Lets use VIM to improve our program

4.12.4. EMACS

https://en.wikipedia.org/wiki/Emacs

- EMACS 1976 David A. Moon and Guy L. Steele Jr
- GNU EMACS ~1985 Richard Stallman
- It has graphical versions but we are going to stick with the terminal mode
- It is a little more recognizable as an editor as it does have menus (`F10` to access menu bar)
  - but it is still pretty cryptic
- Its defining feature is its extensibility
  - no surprise it is built around a programing language. Elisp (a lisp derived language)
  - thousands and thousands of packages
  - extremely configurable
  - lot and lots of programmer tools and support – often used as an IDE unto itself
    - generally I am going shy away from using it this way in class
- all its extensibility and external packages can make emacs very complex
- lots and lots of key-stroke combinations to do stuff
  - easiest to have a cheat sheet
  - use menu system and take note of keyboard shortcuts listed
  - learn to use complex built in help system

- I grew up on emacs
- lets `cd ~/bin` so we start with this as our working directory when we are in vim
- `emacs` or `emacs findfuncs`
- open file
- break the lines up and add a comment

```
#!/bin/bash
# this is our crude bash script to try and find all python functions
# in all the .py files within a directory tree
find ~/kornia -name "*.py" | while read file
do
  grep "def " $file
done
```

- lets add line numbers : `cat -n $file | grep def`
  - exit and demo `cat -n`
- add to loop
  - start with a bug `cat-n $file`
  - exit run

- lets start builing out our IDE: use a second terminal test
  - NEW Terminal
    - set prompt `export PS1='DEBUG$ '` to help us stay organized
  - ET: add `set -x`; DT: run ; ET: add `set -e`
  - ET: edit fix
  - DT: test
- notes:
  - `set -n` : dry run good to check for syntax errors without execution
  - `set -u` : expansion of unset variable is an error good for catching typeos

Good debug tool.
파일을 실행하면서 모든 과정을 print
ex) print Everything from for loop.

- now lets get fancier
- lets extend/modify our code to produce a report of all the names of the functions we find and what files they are in
  - report format: one line per function definition
    - `<function name><space><file>:<space><line number>`
- many ways to go about it but going to try and stick to core shell programming
  - and idioms I have found very useful – learn a few well
- iterate to this code:
  1. add `while read line def func rest; do`
  2. add `echo $func $file: $line`
  3. cleanup function name with `${func%%(*}`
  4. add pipe to sort `| sort`

program name

```
DIR=$1
[[ -z $DIR ]] && echo "Usage: $0 <directory>" && exit 1

find $DIR -name "*.py" | while read file
do
    cat -n $file | grep 'def ' | while read line def func rest; do
    func=${func%%(*}  → creating a function name
    echo $func $file: $line
    done
done | sort
```

```
find ~/kornia -name "*.py" | while read file
do
    cat -n $file | grep 'def '| while read line def func rest; do
        func=${func%%(*}
        echo $func $file: $line
    done
done | sort
```

4.13. Make

- Two things go into any programming development process:
  - modify files
  - running commands that process those files
- Editors are great tools for the first
- but constantly running commands in the terminal is not a great solution for the second
  - There might be many files that we want to run commands on
  - There might be various different commands to run
  - Some commands might depend on on some file
  - etc.

- This is where the `make` utility comes in Tese is a rule to make file.
  - make automates running commands on files when they have changed
    - lets a developer document what "depends" on what
    - and specify what the rules are for generating new/derived files when a dependency has changed
  - the most common use is in the "building" of programs change program to binary file
    - where various commands need to be run on the source files
      - to create the "final" program
    - most IDE either use `make` internal or have some equivalent
  - `make`, however, is a generic tool for automation

- Like the rest of UNIX Make is old, powerful and cryptic npm modern version
  - 1976 Stuart Feldman https://en.wikipedia.org/wiki/Make_(software)
  - There are many variants we are going to focus on GNU Make
    - https://www.gnu.org/software/make/
    - Manual https://www.gnu.org/software/make/manual/make.html
  - Basic syntax and use is not too bad

        - however over the years many features and subtle syntax has been added
- lets use make to automate the use of our `findfunc` command
    - evolving our usage and understanding as we go

4.13.1. Make preliminaries

All of file has a metadata
metadata has "time" of file has been edited.

- Lets revisit the file meta data so we can understand how make detects that something has changed
- Every time a file is modified UNIX updates a field in the meta data of the file called `mtime`
    - every time the file is written `mtime` changes
    - `touch` can update the `mtime` to the current time without changing the contents
- Make uses a simple idea
    - if file 'A' has a modification time that is newer than file 'B'
        - and file 'B' some how depends on file 'A'
            - then 'B' is considered to be out of date
                - any commands/rules that are needed to produce 'B' should be executed
- Note Make is not really looking at the contents of the files only timestamps
    - touch on a 'source' will be sufficient to cause things to be rebuilt

4.13.2. Make Basic Syntax

```
target: dependencies
    rules
```

tab if it is a space then it will give you an error

- must use `<tab>` to indent rule lines
- lets assume every time that the `CHANGELOG.md` is modified we would like to rebuild our report

- BT lets create a project directory : findfunc
- lets put script and makefile in it
- start with

```
funcs.txt: ~/kornia/CHANGELOG.md
    ./findfuncs > $@
```

- lets move towards the following
    1. add target `kornia/CHANGELOG.md:` no dep rule: git clone `https://github.com/kornia/kornia.git`
    2. modify our findfuncs to take directory as a paramter and a usage if empty
    3. progressively add Make variables
    4. add a clean target and .PHONEY
    5. add a funccount.txt target dep: `funcs.txt` rule: `wc -l $< > $@`
    6. add `all` target with `funccount.txt` as dependency
    7. add `torchfuncs.txt` target dep: `funcs.txt` rule: `grep 'torch' $< > $@`
- makefile

```
DIR=kornia
REPO=https://github.com/kornia/kornia.git

.PHONEY: clean all

all: funcscount.txt torchfuncs.txt

funcs.txt: ${DIR}/CHANGELOG.md
	./findfuncs ${DIR} > $@

funcscount.txt: funcs.txt
	wc -l $< > $@

# add this after so that we can see incremental running of rules
torchfuncs.txt: funcs.txt
	grep 'torch' $< > $@

kornia/CHANGELOG.md:
	git clone ${REPO}

clean:
	-rm funcs.txt funcscount.txt torchfuncs.txt

distclean:
	-rm -rf ${DIR}
```

- findfuncs will need to be modified to add an argument

```
#!/bin/bash
# this is our crude bash script to try and find all python functions
# in all the .py files within a directory tree

#set -n  # -n dry run  good for syntax checking (without running any commands)
#set -x  # tracing of simple commands on
#set -e  # exit script on any fail return code
#set -u  # error if expanding unset variable ... goood for catching typeos

DIR=$1
[[ -z $DIR ]] && echo "USAGE: $0 <directory>" && exit -1

find $DIR -name "*.py" | while read file
do
    cat -n $file | grep 'def '| while read line def func rest; do
        func=${func%%(*}
        echo $func $file: $line
    done
done | sort
```