Week 10 Review

5 December 2018

CS 35L Lab 4

Jeremy Rotman

Announcements

- Assignment #9 is due Friday by **11:55pm**
  - **LATE ASSIGNMENTS WILL NOT BE ACCEPTED**
- Remember to submit both your slides and report to CCLE for assignment #10
  - Both of these should be PDF files
  - **LATE ASSIGNMENTS WILL NOT BE ACCEPTED**

Command Line Interface VS Graphical User Interface

Steep Learning Curve	Intuitive
Pure Control	Limited Control
Cumbersome Multitasking	Easy Multitasking
Speed	Limited by pointing
Convenient Remote Access	Bulky Remote Access

Unix File System

- Everything is a file
  - This includes directories (folders) and devices
- Or a process
  - Processes are executing programs
- Files are organized in a tree hierarchy
  - “ / ” - the root directory
    - The topmost directory of the tree
  - “ ~ ” - the home directory
    - User specific
  - “ . ” - the current directory
  - “ .. ” - the parent directory

Moving Around

- pwd
  - print working directory
  - Print the path to the directory you are currently in
- cd
  - change directory
  - Moves the working directory to the specified directory
- man
  - Will open up a manual for any command
  - For example try man man
  - Hit “q” to exit the manual

Basic File Commands

- mv
  - Move a file
  - Alternatively, it is used to rename files
- cp
  - Copy a file
- rm
  - Remove a file
- mkdir
  - Make a directory

Basic File Commands

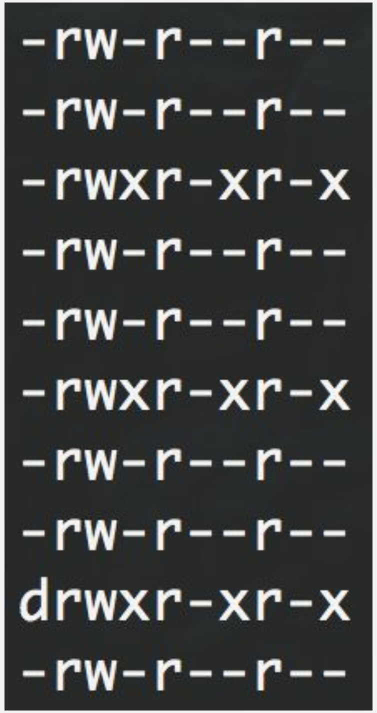
- rmdir
  - Remove an *empty* directory
- ls
  - List the contents of a directory
  - Some useful options
    - -d: lists only directories
    - -a: lists all files, including hidden files
    - -l: lists the long listing which includes file permissions
    - -s: shows size of each file, in blocks

Changing File Attributes

- What are file attributes?
  - The metadata attached to files
- touch
  - Updates the access and modification time to the current time
  - Additionally allows options to specify a time
- ln
  - Creates a link to a file
  - Hard links
    - Point to a file’s physical data
  - Symbolic links (soft links)
    - Point to the file

File Permissions

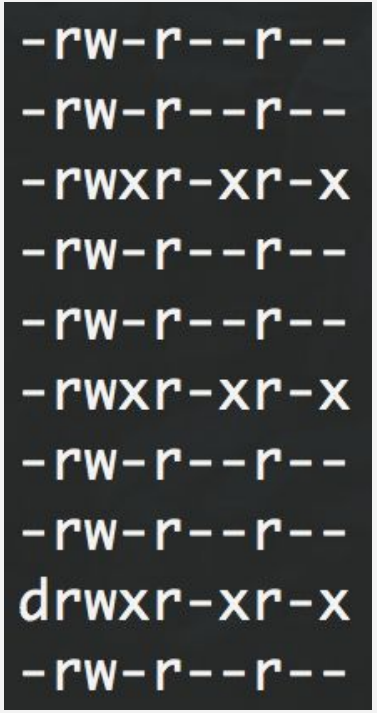
- ➔ Every file has permissions that determine how a user may interact with it
- ➔ These can easily be seen with the command “ ls -l ”
- ➔ The first character is a “d” if the file is a directory
- ➔ The other letters represent:
  - ◆ “r” - read
  - ◆ “w” - write
  - ◆ “x” - executable



```
-rw-r--r--
-rw-r--r--
-rwxr-xr-x
-rw-r--r--
-rw-r--r--
-rwxr-xr-x
-rw-r--r--
-rw-r--r--
drwxr-xr-x
-rw-r--r--
```

File Permissions

- Why are there three groups of three?
- There are permissions for 3 types of users
  - User
    - The owner of the file
  - Group
    - A group that the owner is in
  - Others
    - Anyone else
- In that order



```
-rw-r--r--
-rw-r--r--
-rwxr-xr-x
-rw-r--r--
-rw-r--r--
-rwxr-xr-x
-rw-r--r--
-rw-r--r--
drwxr-xr-x
-rw-r--r--
```

Changing Permissions

- ➔ chmod
  - ◆ Change the mode of the file
- ➔ Can be used in multiple ways
- ➔ chmod [*ugoa*][+-=][*rwx*]
  - ◆ To apply one or more permissions to one or more types of user
  - ◆ Eg. chmod u+x
  - ◆ Symbolic Notation
- ➔ chmod [*0-7*][*0-7*][*0-7*]
  - ◆ To modify all of the permissions in one command
  - ◆ Eg. chmod 754
  - ◆ Octal Notation

Linux Wildcards

When searching for files a few special characters can be very useful

- ?
  - Matches a single occurrence of any character
- *
  - Matches zero or more occurrences of any character
- [ ]
  - Matches any one of the characters between the brackets
  - A “-” can be used for a range of characters

Extra Linux Info

- What is Linux?
  - Linux itself refers to a **kernel**
    - **Kernel:** the core of an OS
    - Allocates time and memory to programs
    - Handles communication between hardware and software
  - The OS is often called GNU/Linux
    - It uses GNU software, and a Linux kernel to make an OS
  - Linux distribution
    - There are many distributions of Linux
    - These mostly vary in what software is included with the OS
  - Linux is open-source
    - The software is openly distributed to be worked on and used

Environment Variables

- Variables that can affect how processes run
- Common ones:
  - **HOME**: path to user's home directory
  - **PATH**: list of directories to search in for command to execute
    - Thus, why you preppended a path in assignment #1
- Changing an environment variable's value

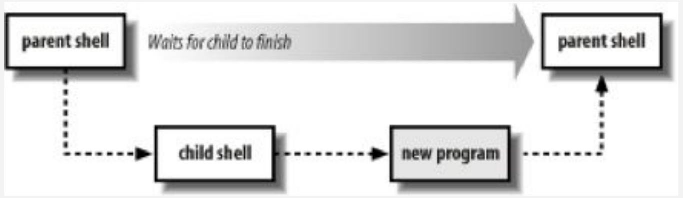
```
export VARIABLE=...
```

What is a shell script?

- ➔ It is a file that holds one or more shell command(s)
- ➔ Why is this useful?
  - ◆ Commands run in succession can be contained in one file
  - ◆ You can introduce conditionals to your shell commands
  - ◆ Clearer variable declarations

The first line

- A shell script spawns a “shell” process to run it



  - Reminder: a shell is a command-line interpreter
  - There are many types of shells
    - E.g. Bash, Bourne shell, C shell, KornShell
  - How do we make sure the correct shell is used in our process?

Quotes

- ➔ Different quotes mean different things
- ➔ Single Quotes: ‘ ’
  - ◆ Literal meaning, do not expand
- ➔ Double Quotes: “ ”
  - ◆ Expand only backticks and $
- ➔ Backticks: ` `
  - ◆ Expand as shell commands

Redirection Refresher

- Programs have 3 streams
  - stdin (0)
    - Contains data going to program
  - stdout (1)
    - Program writes output here
  - stderr (2)
    - Program writes error messages here

Redirection Refresher

- program < file_in
  - Redirects file_in to stdin
- program > file_out
  - Redirects stdout to file_out
- program **2**> file_err
  - Redirects stderr to file_err
- program >> file_out
  - Appends stdout to file_out
- program1 | program2
  - Redirects stdout from program1 to stdin of program2

RegEx

- Allows you to search for patterns rather than a direct search
- Similar to wildcards
  - Note however, that they function slightly differently

Anchors

- ^
  - Match the following regular expression with the beginning of a line or string
- $
  - Match the preceding regular expression with the end of a line or string

Quantifiers

- .
  - Match any single character
- *
  - Match 0 or more of the preceding character
- +
  - Match 1 or more of the preceding character
- ?
  - Match 0 or 1 of the preceding character

Quantifiers

- **$\{n\}$**
  - Match exactly n of the preceding character
- **$\{n,\}$**
  - Match n or more of the preceding character
- **$\{n,m\}$**
  - Match n to m of the preceding character

Bracket

- **[...]**
  - Allows a match to any one of the enclosed characters
- **-**
  - A hyphen within a bracket designates a range, like A-Z

POSIX Bracket expressions

Expression	Meaning
[:alnum:]	Alphanumeric Characters
[:alpha:]	Alphabetic Characters
[:blank:]	Space and Tab Characters
[:cntrl:]	Control Characters
[:digit:]	Numeric Characters
[:graph:]	Nonspace Characters

POSIX Bracket expressions

Expression	Meaning
[:lower:]	Lowercase Characters
[:print:]	Printable Characters
[:punct:]	Punctuation Characters
[:space:]	Whitespace Characters
[:upper:]	Uppercase Characters
[:xdigit:]	Hexadecimal Digits

Parentheses

- Parentheses allow you to apply quantifiers to sequences of characters
    - E.g. (ab)*
- Additionally, parentheses form capturing groups
    - These can be backreferenced later in the regular expression
    - E.g. (ab)c\1c
    - You can only store 9 capturing groups (\1-\9)

BRE vs ERE

- ➔ Basic Regular Expression (BRE) is the standard mode for sed and grep
- ➔ Extended Regular Expression (ERE) is an optional flag you can use with the commands
- ➔ What’s the difference?
  - ◆ BRE tends to take things more literally

BRE vs ERE

- In BRE ‘ ? ’, ‘ + ’, ‘ { ’, ‘ } ’, ‘ (’, and ‘) ’ lose their special meanings
  - They are treated literally
- To use the special meanings, you will either need to use the ERE option, or use ‘ \ ’
  - E.g. \(ab\)\+
- For characters with special meaning, ‘ \ ’ can be used to turn off special meanings of characters
  - Yes, it’s a bit confusing

Searching text with RegEx

- `grep [OPTIONS] PATTERN [FILE...]`
  - Search either FILE(s) or STDIN for the given RegEx PATTERN
  - It will return matching lines
- Useful grep options
  - -E
    - Uses extended regular expressions
  - -F
    - Matches fixed strings

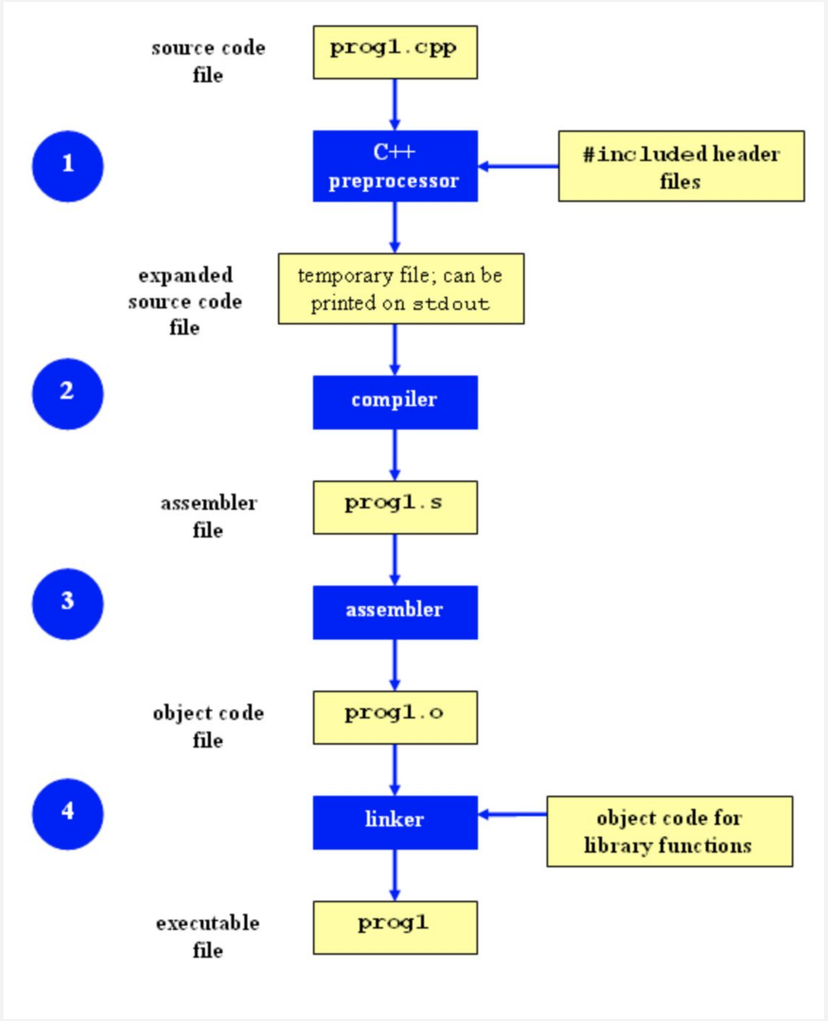
Replacing text with RegEx

- ➔ `Sed SCRIPT [FILE]`
  - ◆ Stream Editor
- ➔ How to construct SCRIPT?
  - ◆ Theoretically many ways, but we will focus on text replacement
    - `'s/`***`regexp`***`/`***`replacement`***`/`***`flags`***`'`
  - ◆ The ‘s’ signifies substitution
  - ◆ Useful flags
    - g - global, replace all matches to ***regexp***, not just the first
    - I - case-**in**sensitive matching
- ➔ E.g.
  - ◆ `sed 's/[Jj]eremy/John/g'`

C++ compilation process

```
g++ -Wall -o prog1 prog1.cpp
```

1. C++ preprocessor copies header, generates macro, and checks defined constants
2. Source code compiled to assembly
3. Assembler code is assembled into object code
4. Object code is linked with object code files for library functions, producing an executable



Compilation

- What does it mean when we compile code?
  - Essentially translating code that we can read to code that a computer can quickly read and execute
- So what do other languages do?

Interpreters

- ➔ How does an interpreter differ from a compiler?
  - ◆ Rather than one compile step to translate the entire source, an interpreter reads the source and translates line by line
- ➔ Advantages and Disadvantages
  - ◆ Programs run slower
    - But you can usually run them sooner
  - ◆ Simpler, and probably more intuitive
  - ◆ More portable
  - ◆ But it really is much slower
    - There is a lot of overhead
    - Added to extra time to actually call variables

Makefile Example

```
CC=g++
CFLAGS=-Wall -g
all: BandN
BandN: coffee.o book.o bookstore.o
    $(CC) $(CFLAGS) -o BandN coffee.o book.o bookstore.o
coffee.o: coffee.cpp coffee.h
    $(CC) $(CFLAGS) -c coffee.cpp
book.o: book.cpp book.h
    $(CC) $(CFLAGS) -c book.cpp
bookstore.o: bookstore.cpp book.h coffee.h
    $(CC) $(CFLAGS) -c bookstore.cpp
clean:
    rm -f coffee.o book.o bookstore.o BandN
```

Make commands

- ./configure
  - The configure script checks to make sure the machine is ready to install software
  - Checks for proper dependencies
  - Usually generates the Makefile from an unfinished file
- make
  - Requires a Makefile
  - Builds the software
- make install
  - Runs the install label in the Makefile
  - Copies the built software to the system’s directories

How to read diff output

- When you run normal diff
  - Outputs differences without context
- < indicates the lines that are unique to file 1
- > indicates the lines that are unique to file 2

```
change-command
< from-file-line
< from-file-line
---
> to-file-line
> to-file-line
```

How to read diff output

- ➔ change-command has one of 3 formats
  - ◆ *“lar”*
    - Add lines in range *r* from the 2nd file, starting after line *l* in the 1st file
  - ◆ *“fct”*
    - Change lines in range *f* from the 1st file to range *t* from the 2nd file
  - ◆ *“rdl”*
    - Delete the lines in range *r* from the 1st file, line *l* is where they would have appeared in the 2nd file if not deleted

```
change-command
< from-file-line
< from-file-line
---
> to-file-line
> to-file-line
```

Unified Diff Output

```
--- from-file from-file-modification-time
+++ to-file   to-file-modification-time

@@ from-file-line-numbers to-file-line-numbers @@
Line-from-either-file
Line-from-either-file
+Line-added-to-first-file
-Line-removed-from-second-file
Line-from-either-file
```

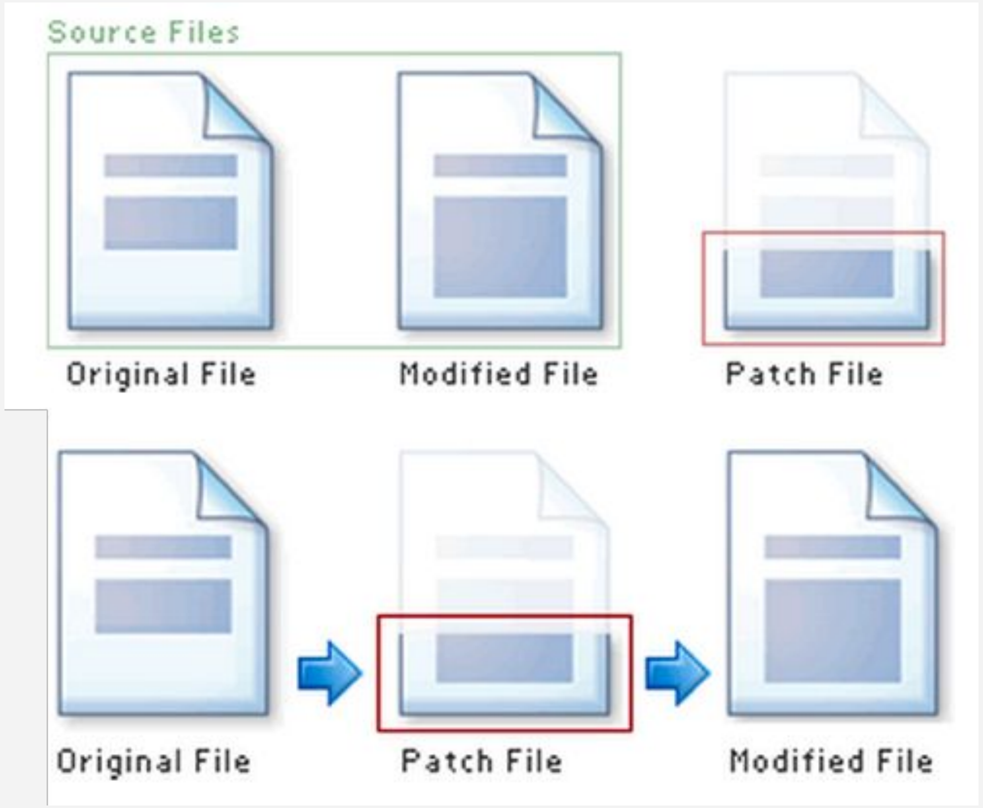
Unified Diff Output

- ➔ Note that the first file is “-” and the second file is “+”
- ➔ Another way to differentiate is to match the symbol in the header to lines
  - ◆ The file that has +++ in front of it in the header, has the lines with a + in front of them

Patching

- ➔ You may be very familiar with the term if you play video games
- ➔ But what exactly is a patch?
  - ◆ A piece of software designed to fix problems with or update a computer program
- ➔ A Patchfile is just a diff file that includes the changes made to a file
- ➔ The patch command can be used with the diff file to add changes

Applying a Patch



The patch command

- **`patch -p`**`*num* < *patchfile*`
  - The -p option allows you to strip the smallest prefix containing *num* leading slashes
  - In particular this relates to filenames within the patchfile
  - If you have a different directory for your filenames, this is important to use
  - E.g. if you had the filename /u/home/j/jeremy/src/coreutils
    - -p0 gives /u/home/j/jeremy/src/coreutils
    - -p1 gives u/home/j/jeremy/src/coreutils
    - -p5 gives src/coreutils

Optparse Actions: Store

- The most basic and probably most useful action
- It is also the default action
  - Meaning you don’t technically need to specify it
- This includes 3 arguments in the add_option function
  - action=“store”
    - Declares the action as store
  - type=“string”
    - Declares the type of argument being stored as a string
    - String is default
  - dest=“var_name”
    - Indicates the variable you want to store the argument as

Optparse Actions: Store

So an example may look like:

```
parser.add_option(“-f”, “--file”,
   action=“store”, type=“string”, dest=“filename”)
```

Optparse Actions: Booleans

- If you want an option that is simply a flag to turn things on or off, there is a different set of actions similar to store
  - store_true
  - store_false
- These will still need to be stored in a variable, but the type is not important

Optparse Actions: Booleans

So for example:

```
parser.add_option("-v", action="store_true",
   dest="verbose")

parser.add_option("-q", action="store_false",
   dest="verbose")
```

Default Values for Options

- ➔ Everytime you add an option, you can include a default value
- ➔ Say for example, we want the option for the script to be verbose, but want it to be quiet by default

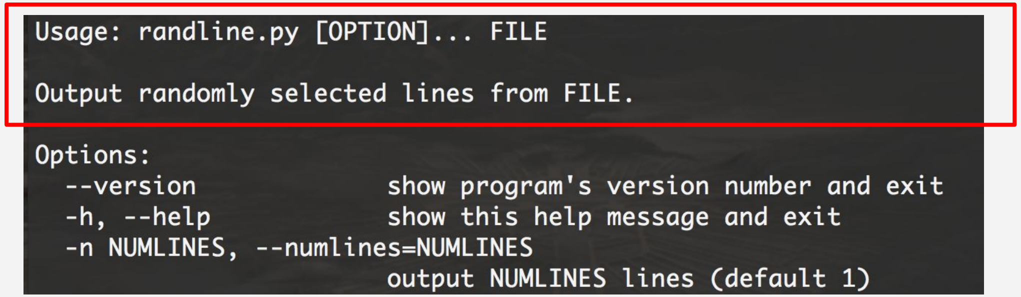
```
parser.add_option("-v", action="store_true",
   dest="verbose", default=False)
```

Program Usage

- When creating your OptionParser, you can specify a program’s usage message
- What is a usage message?

Program Usage

- ➔ When creating your OptionParser, you can specify a program's usage message
- ➔ What is a usage message?
  - ◆ The line at the top of the help file describing how to run the program
  - ◆ Additionally, it might include a description of what the program does



```
Usage: randline.py [OPTION]... FILE

Output randomly selected lines from FILE.

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -n NUMLINES, --numlines=NUMLINES
                        output NUMLINES lines (default 1)
```

Program Usage

For example this creates the usage message in randline.py

```
usage_msg = """%prog [OPTION]... FILE

Output randomly selected lines from FILE"""
parser = OptionParser(usage=usage_msg)
```

Option Help Messages

- In addition to the usage at the top, the help display also showed help messages for all of the options that you could use
- This message must be defined in the add_option function

Option Help Messages

Going back to our verbose example before:

```
parser.add_option(“-v”, action=“store_true”,
   dest=“verbose”, default=False,
   help=“Print out extra information about
   the program while running”)
```

GDB

- Compiling for GDB
  - `gcc [other flags] -g src.c -o src`
  - The -g option links debugging symbols into the program
    - This makes GDB more effective
- Entering GDB
  - Now that you have your executable, you can enter the debugger with it
  - **`gdb`** *`src`*
  - OR
    - **`gdb`**
    - `(gdb)` **`file`** *`src`*

Breakpoints

- ➔ Once set, you can rerun the program and it will stop when it reaches the breakpoint
- ➔ You can have many breakpoints
  - ◆ `(gdb) info breakpoints | break | br | b`
    - Shows the list of breakpoints and information about where they are set
    - Will also include the breakpoint number which is important for other commands that deal with breakpoints

Breakpoints

- `(gdb)` **`delete`** *`bp_number`*
  - Deletes the breakpoint, or range of breakpoints if given range
- `(gdb)` **`disable`** *`bp_number`*
  - Disables, but doesn't remove, the breakpoint(s)
- `(gdb)` **`enable`** *`bp_number`*
  - Enables previously disabled breakpoint(s)
- If no arguments are given, it affects all breakpoints
- `(gdb)` **`ignore`** *`bp_number iterations`*
  - Pass over a breakpoint without stopping *iterations* number of times

After the Breakpoint

- There are 4 operations to continue after the breakpoint
  - **c** or **continue**
    - Debugger will continue to run until it hits another breakpoint, error, or finishes running
  - **s** or **step**
    - Debugger will continue to next line
    - Steps into function calls
  - **n** or **next**
    - Debugger will continue to next line in current stack frame
    - Steps over function calls
  - **f** or **finish**
    - Debugger will continue until the current function returns to the function that called it

Watchpoints

- ➔ Watchpoints are attached to variables and will stop the program whenever an action is taken on the variable
- ➔ `(gdb)` **`watch`** *`expression`*
  - ◆ Set watchpoint on *expression*
  - ◆ Stop when the value of *expression* is changed
  - ◆ Prints old and new values
- ➔ `(gdb)` **`rwatch`** *`expression`*
  - ◆ Stops when the value of *expression* is read

Pointers

- ➔ Variables that store memory addresses
  - ◆ Generally they store the memory address at which a variable is stored
- ➔ Declaration
  - ◆ *<var_data_type> *<ptr_name>*

```
int* ptr;           // declares ptr as a pointer to an int variable
int var = 6;        // defines an int variable var
ptr = &var;         // sets the pointer equal to the memory address of var
```

- ➔ Dereference (access the value stored in memory) with *

```
*ptr = 15;          // sets var’s value to 15
```

More Pointers

- Can a pointer point to a pointer?
  - Yes, they can
  - Pointception

```
char c = ‘Z’
char* cPtr = &c
char** cPtrPte = &cPtr
```

Function Pointers

- Functors
- Allows you to pass a function into another function
- Why is this useful?
  - It allows you to create a more generalized function
  - Conditionals can allow you to use a different function within the same function

Functors

- ➔ Declaration

```
double (*func_ptr) (double, double);
func_ptr = &pow;        // func_ptr now points to pow()
```

- ➔ Usage

```
double result = (*func_ptr)(1.5, 2.0);
```

C Structs vs. C++ Classes

C Structs	C++ Classes
No member functions	Member functions
No access specifiers	Access specifiers that are private by default
No defined constructors	Must at least have default constructor

Dynamic Memory

- ➔ C allows you to allocate memory at run time
  - ◆ This gets allocated on the heap

- ➔ void* **malloc**(size_t *size*);
  - ◆ Allocates *size* bytes and returns a pointer to the allocated memory
- ➔ void* **realloc**(void* *ptr*, size_t *size*);
  - ◆ Changes the size of the memory block pointed to by *ptr* to *size* bytes
- ➔ void **free**(void* *ptr*);
  - ◆ Frees the block of memory pointed to by *ptr*

Reading and Writing Characters

- int **getchar**();
  - Returns the next character from stdin
- int **putchar**(int *character*);
  - Writes *character* to the current position in stdout
- int **fprintf**(FILE* *stream*, const char* *format*, …);
  - Print the c string *format* to *stream*
    - Stream is either a pointer to a file, or one of stdin, stdout, stderr
- int **fscanf**(FILE* *stream*, const char* *format*, …);
  - Read the c string *format* from *stream*

Formatted I/O

- fprintf and fscanf rely on file pointers and formatted strings
- File pointers
  - Generated by opening files
  - file* fp = fopen(“file.txt”, “r”)
    - Fp is now a pointer to the opened file file.txt with read permission
- Formatted String
  - Allows a string to be followed by variables to be placed into a string
  - fprintf(fp, “This class, %s, has %i students”, class_name, n_students)
  - For fscanf, the formatted string will allow you to place separate pieces of the string into variables

Processor Modes

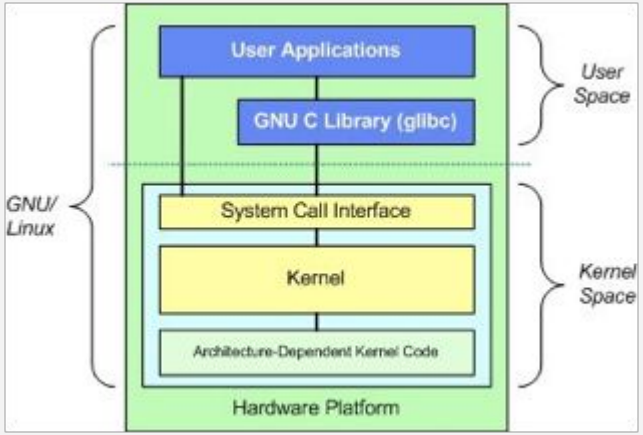
- The CPU (in Linux) has two distinct modes of operation
  - Kernel mode
    - Unrestricted access
    - Can execute any instruction, and reference any memory address
    - Assumes it is running trusted software
  - User mode
    - Non-privileged access
    - Cannot directly access hardware
    - Must use a system call to perform privileged instructions

Why Dual-Mode Operation?

- I/O protection
  - Input/output cannot be directly controlled by user-created code
- Memory protection
  - User mode only has access to a set partition of memory
  - The user is not allowed to access memory addresses that define these bounds, or other addresses outside their partition
- CPU Protection
  - User mode is not allowed to change things related to the OS's scheduler or timer

Trusted Software

- ➔ What is the trusted software that can be run in kernel mode?
  - ◆ Software in the kernel space
    - Cannot be changed from the outside
    - Implements protection mechanisms
- ➔ System call interface bridges the gap between User Mode and Kernel Mode
  - ◆ User processes can execute privileged operations through the interface

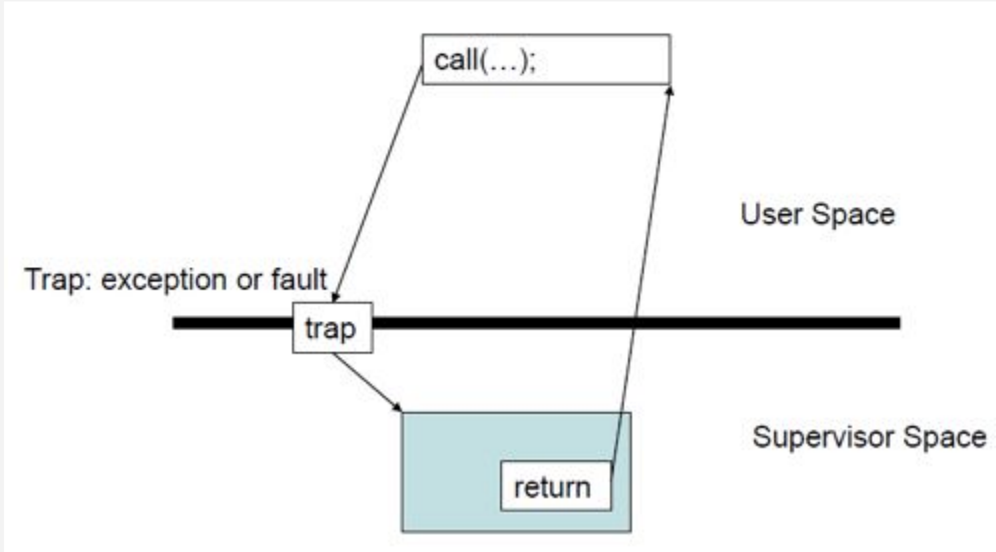


System Calls

- ➔ Used by user-level processes to request service from kernel
- ➔ Changes CPU’s mode from User to Kernel
- ➔ Part of the Kernel of the OS
- ➔ Verifies User should be allowed to do operation, then handles operation
- ➔ The only way that a user program can perform privileged operations

System Calls

- When a system call is made
  - The program being executed is interrupted
  - Control is passed to the kernel
  - If the operation is valid
    - Kernel performs it
  - Else
    - Throw an exception



System Calls

- Overhead
  - System calls include a lot of overhead
  - Many things that must be done
    - Process interrupted and computer saves its state
    - OS takes control of CPU and verifies validity of operation
    - OS performs requested action
    - OS restores saved context, switches to user mode
    - OS gives control of the CPU back to user process

C Library Functions

- ➔ There are functions that are part of the standard C library that do the same thing
  - ◆ getchar & putchar are similar to read & write (standard I/O)
  - ◆ fopen & fclose are similar to open & close (file I/O)
- ➔ These C functions then make system calls of their own
- ➔ What is the advantage?
  - ◆ Library functions make fewer system calls
  - ◆ This reduces the amount of overhead
  - ◆ Efficiency :D

Unbuffered vs. Buffered I/O

- Unbuffered
  - Every byte is read/written by the kernel through system call
- Buffered
  - Collect as many bytes as possible (in a buffer) and read/write them all in one system call
- Buffered I/O reduces overhead because you don’t have to switch mode for every byte
  - Thought exercise: when might you want to use unbuffered I/O?
  - Also, you must still always be prepared for buffer overflow attacks

C Option Parsing

- int getopt(int argc, char* const* argv, const char* options)
  - Asks for number of arguments, the arguments, and the options
  - Options holds the flags you might expect to receive
    - It is a string of character flags, e.g. “abc” means options a, b, and c
    - By adding a ‘:’ after a character it implies that character should have an argument after it, e.g. “a:bc” means option a has an argument
- int optopt
  - If getopt receives an unknown option, it sets this to be that option
- char* optarg
  - Variable set by getopt to point to the argument of the option

Bash Options

- ➔ Relatively similar to C getopt
- ➔ getopts OPTSTRING VARNAME [ARGS…]
  - ◆ VARNAME is the variable getopts gets stored in
  - ◆ OPTSTRING is essentially the same as the option string in C
  - ◆ One major difference
    - A colon at the beginning of the string changes the error reporting mode
    - Default is verbose, a colon at the beginning switches it to silent
  - ◆ Verbose vs. Silent
    - Verbose
      - ○ Invalid option: VARNAME is set to ?, OPTARG is unset
    - Silent
      - ○ Invalid option: VARNAME is set to ?, OPTARG is set to the invalid option

Parallelism

- Executing several computations simultaneously to gain performance
- Two primary forms of parallelism
  - Multitasking
    - Several processes are scheduled in an alternating pattern
    - Potentially, a simultaneous pattern if on a multiprocessing system
  - Multithreading
    - The same job is split into logical pieces (threads)
    - Threads may be executed simultaneously in a multiprocessing system

Parallelism

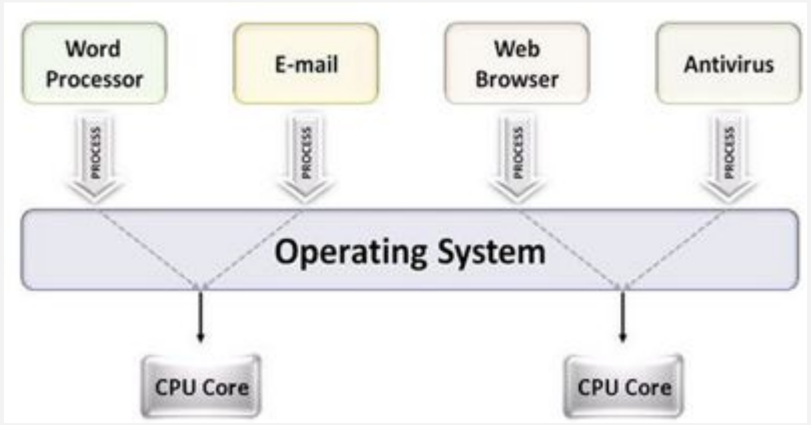
- Executing several computations simultaneously to gain performance
- Two primary forms of parallelism
  - Multitasking
    - Several processes are scheduled in an alternating pattern
    - Potentially, a simultaneous pattern if on a multiprocessing system
  - Multithreading
    - The same job is split into logical pieces (threads)
    - Threads may be executed simultaneously in a multiprocessing system

Threads

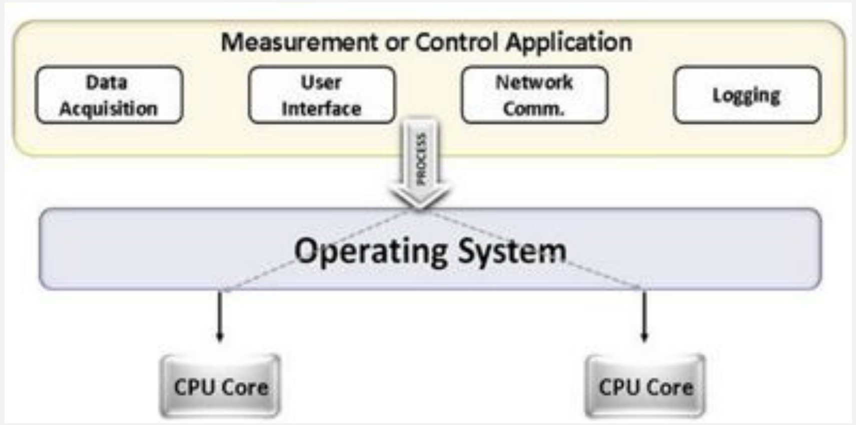
- ➔ A flow of instructions, or path of execution within a process
- ➔ Smallest unit of process scheduled by the OS
- ➔ A process consists of at least one thread

Multitasking vs. Multithreading

<u>Multitasking</u>



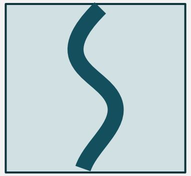
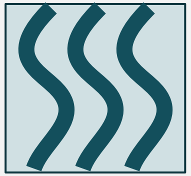
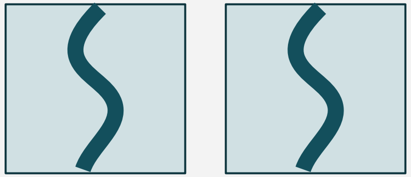
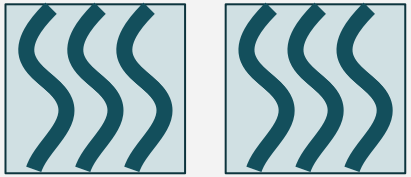
<u>Multithreading</u>



Using Both

Multithreading

Multitasking

	No	Yes
No	 One process One thread	 One process Multiple threads
Yes	 Multiple processes One thread each	 Multiple processes Multiple threads in each

pthread_create

- ➔ Create a new thread and make it executable
- ➔ Can be called any number of times from anywhere in the code
- ➔ Return
  - ◆ Success: 0
  - ◆ Failure: Error Code

pthread_join

- Makes originating thread wait for the completion of all of its spawned reads
- Without join, the originating thread would exit as soon as it’s done with the jon
  - A spawned thread can be aborted even if it is in the middle of its chore
- Return
  - Success: 0
  - Failure: Error Code

Linking

- ➔ What does the linker actually do?
  - ◆ The linker collects procedures and links together the object modules into one executable program
- ➔ Why use linking instead of a single huge program?
  - ◆ Efficiency
    - Only need to recompile the external functions
    - Don’t need to recompile the entire huge program

Static Linking

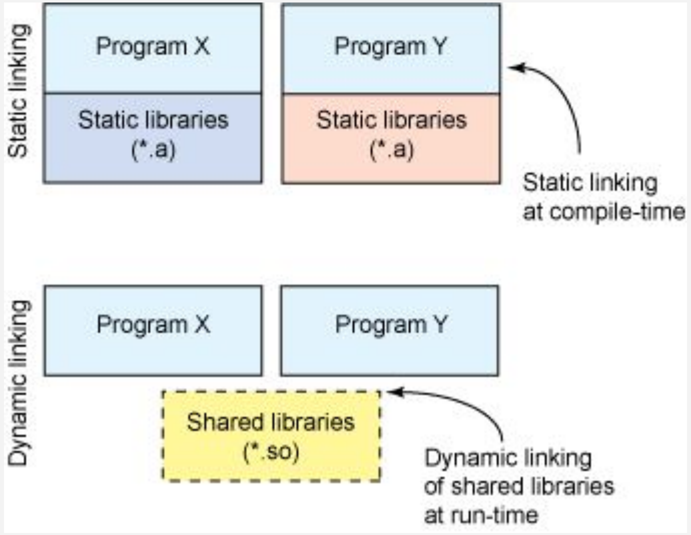
- ➔ Carried out once to produce an executable file
- ➔ If static libraries are called, the linker copies all modules referenced by the program to the executable
- ➔ Typically denoted by the .a extension

Dynamic Linking

- Allows a process to add, remove, replace or relocate object modules during execution
- If shared libraries are called
  - Only copy a little reference information when the executable file is created
  - Complete the linking during loading time or running time
- Typically denoted by the .so file extension
  - .dll on windows

Static vs. Dynamic Linking

- Static Linking
  - Each program has its own complete copy of the library
- Dynamic Linking
  - Each program shares a reference to the shared library
  - The library is loaded at execution



Advantages and Disadvantages of Dynamic Linking

- Advantages
  - The executable is smaller
  - If the library changes, the code referencing it does not need to recompile
  - Multiple programs can access the .so file at the same time
- Disadvantages
  - Performance hit
    - Objects need to be loaded, addresses need to be resolved, etc.
  - Missing libraries
  - Varying versions of libraries

GCC Flags

- -fPIC
  - Compiler directive to generate Position Independent Code
- -l*library*
  - Searches for the file lib*library.a*
  - If not given -L, it defaults to the /usr/lib path
- -L
  - At compile time, search for the library from this path
- -Wl,*option*
  - passes *option* to the linker
  - For multiple options use more commas

GCC Flags

- -rpath=.
  - At runtime, find .so from this path
- -c
  - Generate object code from c code
- -shared
  - Produce a shared object which can then be linked with other objects to form an executable

Attributes of Functions

- ➔ Used to declare certain things about functions called in your program
- ➔ Also used to control memory placement, code generation options or call/return conventions within the function being annotated
- ➔ Introduced by the attribute keyword on a declaration, followed by an attribute specification inside double parentheses

Attributes of Functions

- ➔ __attribute__((constructor))
  - ◆ Will make the function run before main
  - ◆ Runs when dlopen is called
- ➔ __attribute__((destructor))
  - ◆ Will make the function run after main
  - ◆ Runs when dlclose() is called
- ➔ E.g.

```
__attribute__((constructor))
void run_before(void) {
        printf(“called first\n”);
}
```

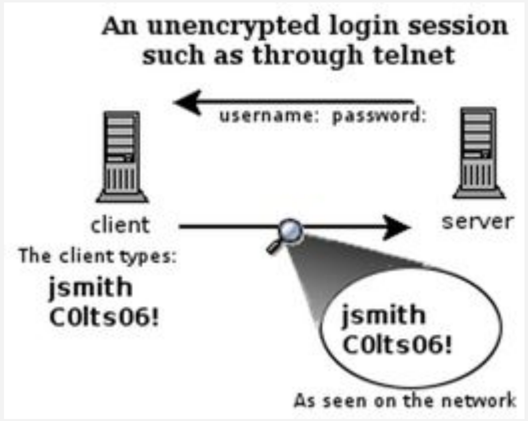
Communicating Over the Internet

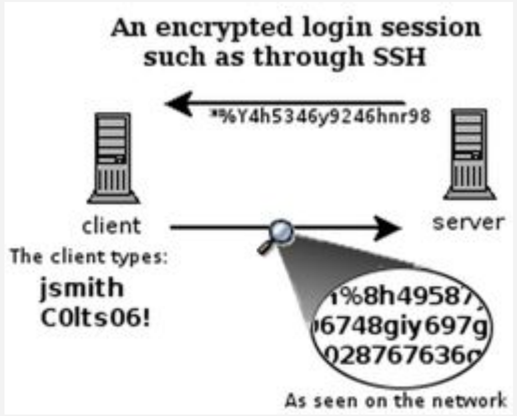
What guarantees do we want?

- Confidentiality
  - Message secrecy
- Data integrity
  - Message consistency
- Authentication
  - Identity confirmation
- Authorization
  - Resource access rights specification

SSH

- ➔ Secure Shell
- ➔ Used to remotely access shell
- ➔ Successor of telnet
- ➔ Encrypted and better authenticated session

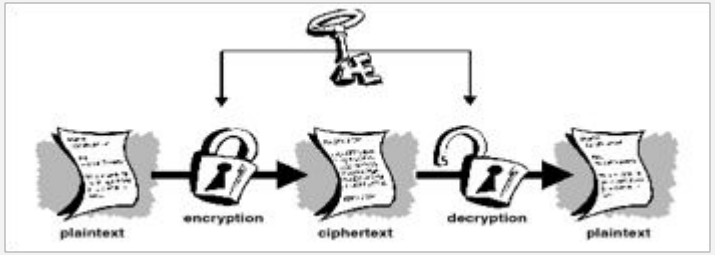


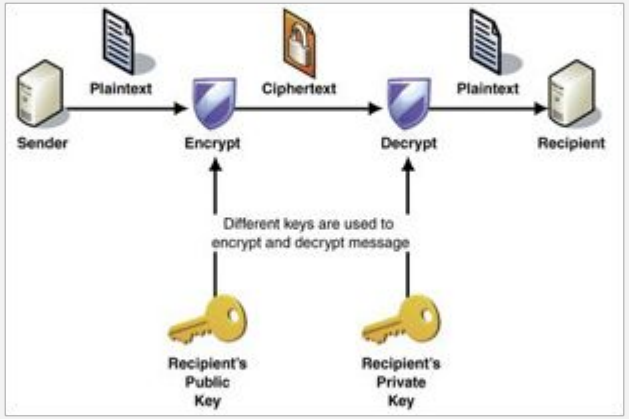


Encryption Types

- Symmetric Key Encryption
  - shared/secret key
  - Key used to encrypt is the same as key used to decrypt
- Asymmetric Key Encryption
  - public/private
  - 2 different, but related keys, public and private key
    - Private key cannot be derived from the public key
  - Data encrypted with public key can only be decrypted with private key
  - Public key can be seen by anyone
  - **Never** publish private key

Symmetric vs. Asymmetric





Symmetric

Asymmetric

SSH Protocol

Client SSH's to remote server:

- ➔ If this is the first time:
  - ◆ ssh does not know the host
  - ◆ Shows hostname, IP address, and fingerprint of the server's public key so you can confirm the correct host
  - ◆ If the client accepts, connection, it saves the public key to its known hosts
    - ~/.ssh/known_hosts

SSH Protocol

- The next time, if the host's public key does not match what was already saved
  - Client encrypts a message with the public key
  - The host decrypts this with its private key to prove that it is the real host
- Once the host is verified
  - Host and client agree on a symmetric encryption key
    - For the session
  - All messages between the host and client are encrypted and decrypted with the session key

SSH Protocol

- User Authentication
  - Password authentication
    - User is prompted for password
  - Key-based authentication
    - Generate key-pair
    - Copy the public key to the server
    - Server authenticates client if it can demonstrate it has the private key
    - Private key can have a passphrase attached
      - But this forces you to enter passphrase every time the private key is used

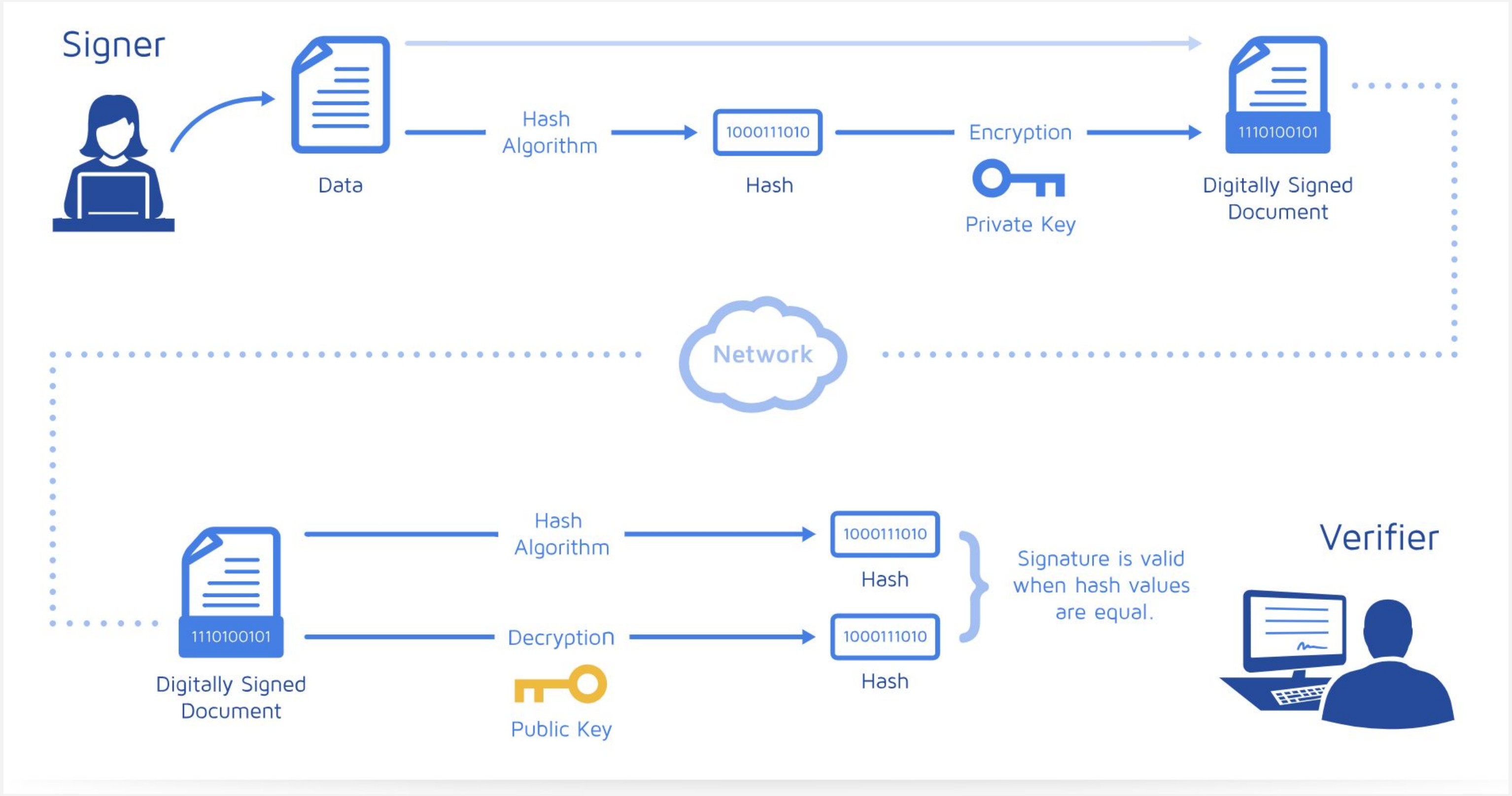
ssh-agent

- ➔ Program that works with OpenSSH that provides a secure way of storing private key
- ➔ `ssh-add` prompts user for passphrase once and adds to the list maintained by `ssh-agent`
- ➔ Once passphrase is added to `ssh-agent`, user will not be prompted for it again when using SSH
- ➔ OpenSSH will talk to ssh-agent and retrieve private key automatically

Digital Signatures

- Electronic stamp or seal
  - Certifies and timestamps the document
  - Acts like a handwritten signature, but also adds more
- Digital Signatures are tamper-proof
  - This ensures data integrity
  - If the file is changed after being signed
    - The signature cannot be verified

Digital Signatures



Detached Digital Signatures

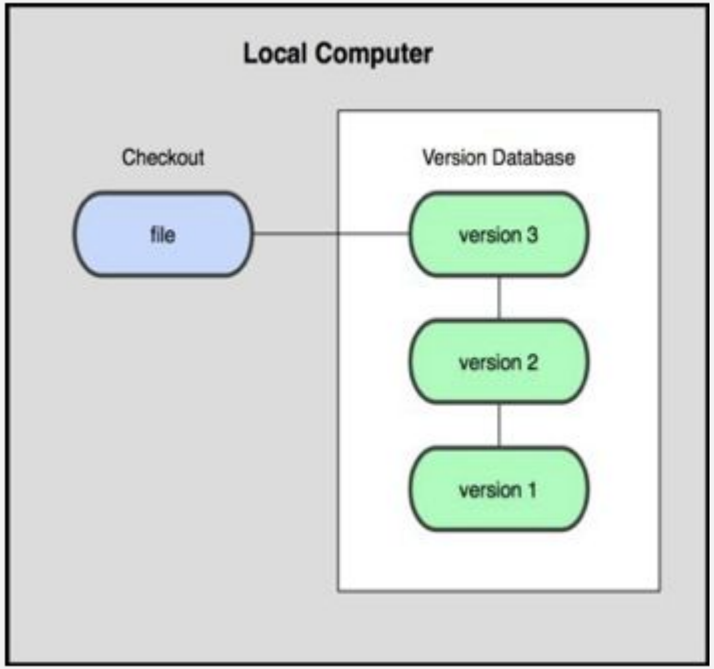
- A digital signature must compress the original document
  - This is not always ideal
  - A clearsigned document is an option
    - But the document must still be edited in some way
- What to do if you are signing a tarball?
- A detached digital signature is stored and transmitted in a file that is separate from the original file
  - Both must be used in the verify command to verify the signature

Source/Version Control

- Track changes to code and other files related to the software
- Track entire history of the software
- Various Version Control Software (VCS) to do this
  - Git
  - Subversion
  - Perforce

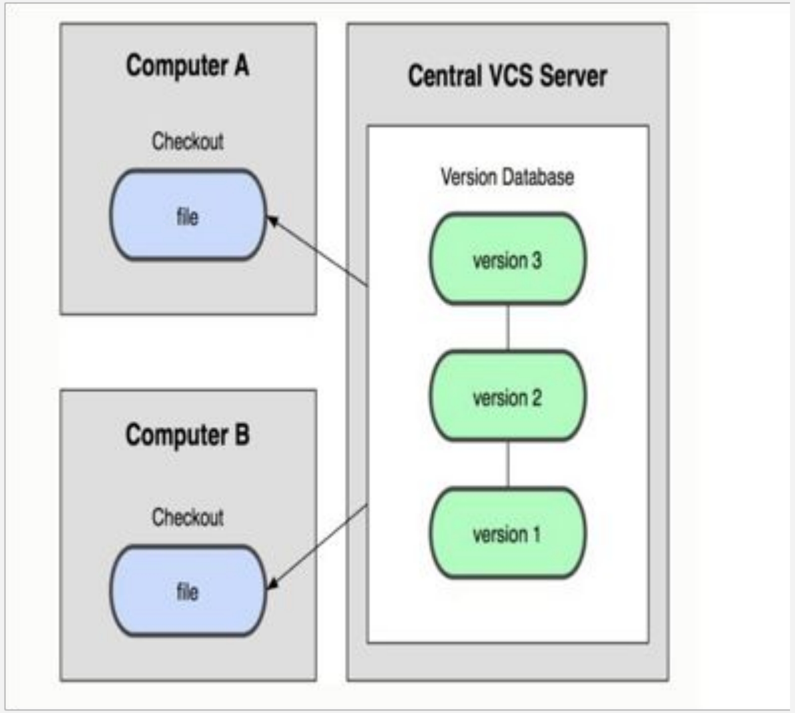
Local VCS

- ➔ Different versions of the software are in different folders in the local machine
- ➔ No server
- ➔ Other users should copy via disk/network



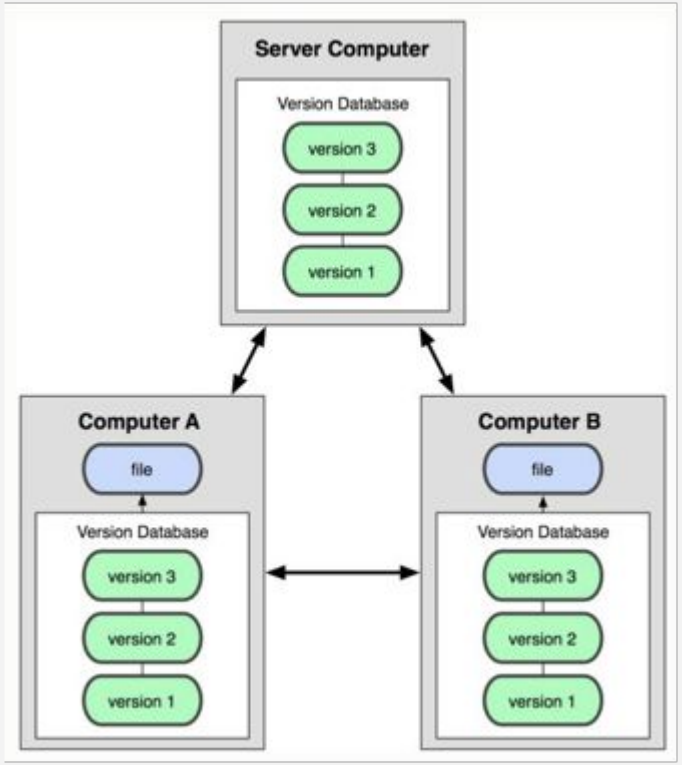
Centralized VCS

- ➔ Version history on a central server
- ➔ Users get working copy of files
- ➔ Changes must be committed to the server
- ➔ All users can get changes



Distributed VCS

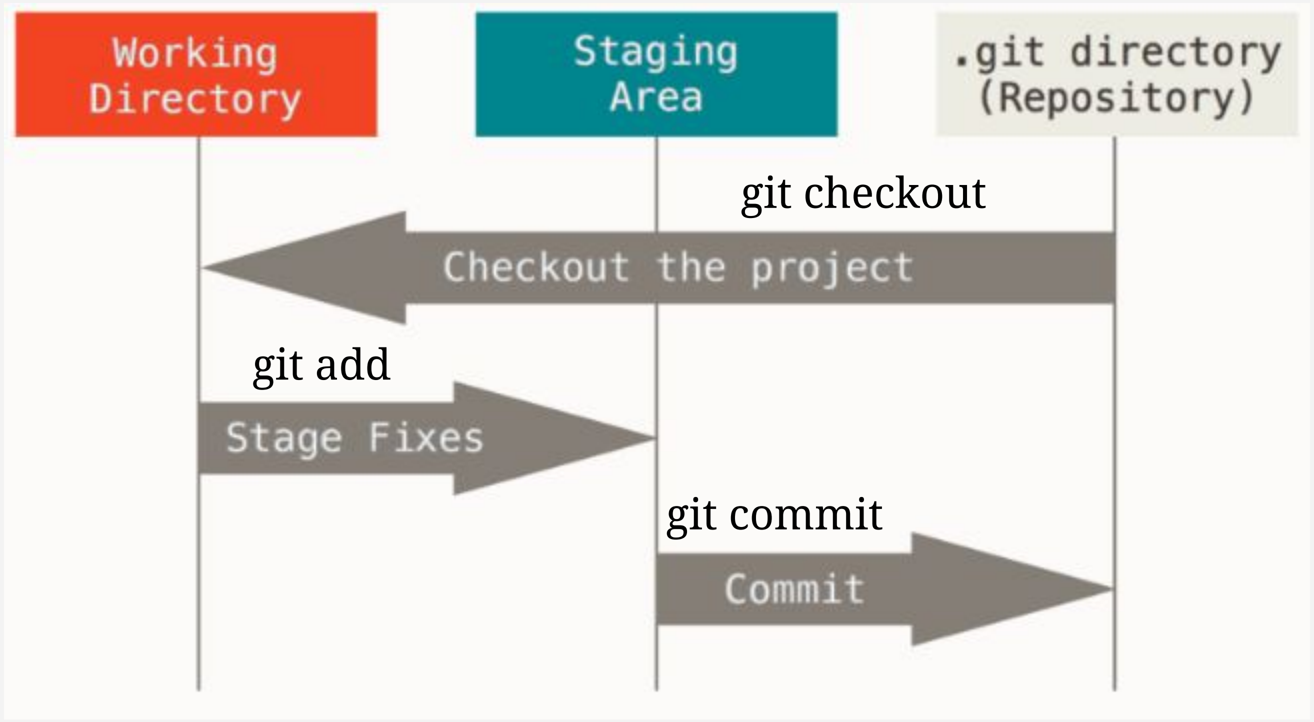
- ➔ All users mirror the entire version history
- ➔ Users have version control at all times
- ➔ Changes can be communicated between users
- ➔ Git is distributed



Useful Terms

- ➔ Repository (repo)
  - ◆ Files and folders related to the software code
  - ◆ Full history of the software
- ➔ Working copy
  - ◆ Copy of the software’s files in the repository
- ➔ Check-out
  - ◆ To create a working copy of the repository
- ➔ Check-in / Commit
  - ◆ Write the changes made in the working copy to the repository
  - ◆ Commits are recorded by the VCS

Git States

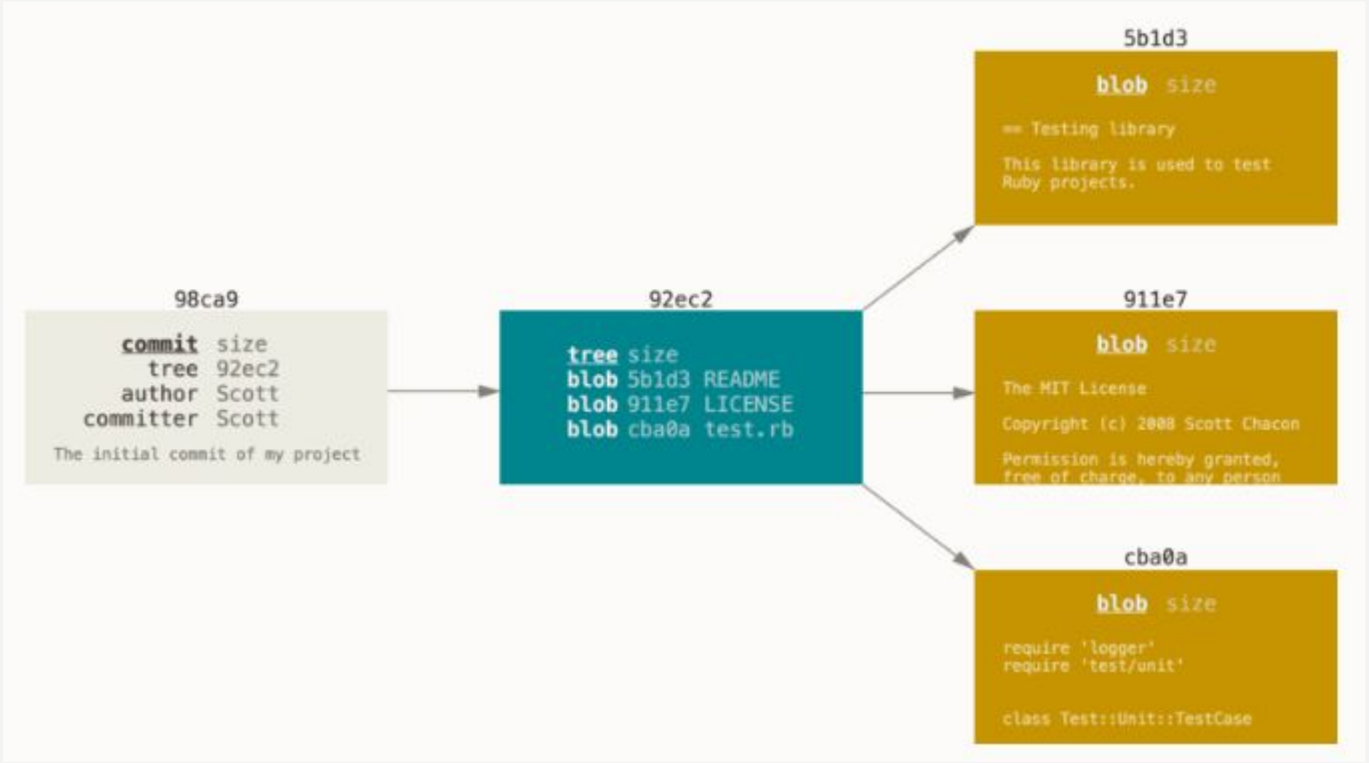

Working Directory
Staging Area
.git directory (Repository)
git checkout
Checkout the project
git add
Stage Fixes
git commit
Commit

Git Repository Objects

- ➔ Blobs
  - ◆ Binary object used to store the contents of each file
- ➔ Trees
  - ◆ The hierarchy between files stored in a repository
- ➔ Commit
  - ◆ Refers to a particular “git commit”
  - ◆ A snapshot of the git tree and blobs in a repository
- ➔ Tags
  - ◆ Essentially attached names for specific commits

Git Repository Objects

A git commit is a snapshot of the repository, a version



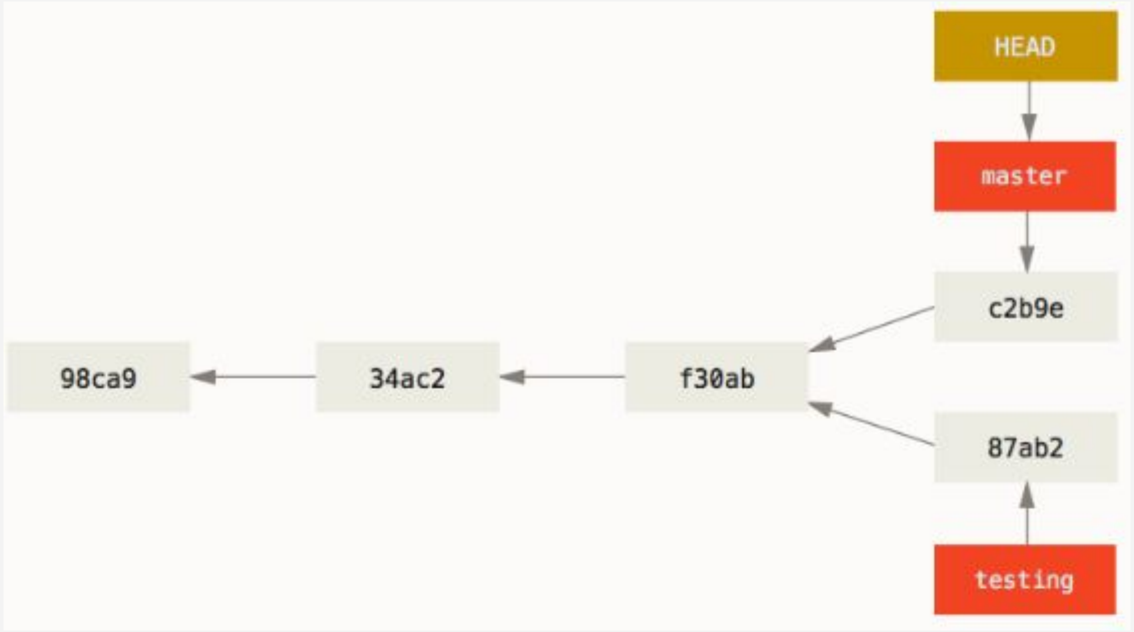
More Terms!

- Branch
  - An active line of development
- Head
  - Named reference to a commit at the tip of a branch
- HEAD
  - Your currently active branch
- Detached HEAD
  - Occurs when you check-out a commit that is not necessarily part of any branch
- Master
  - Default branch

Branching Example

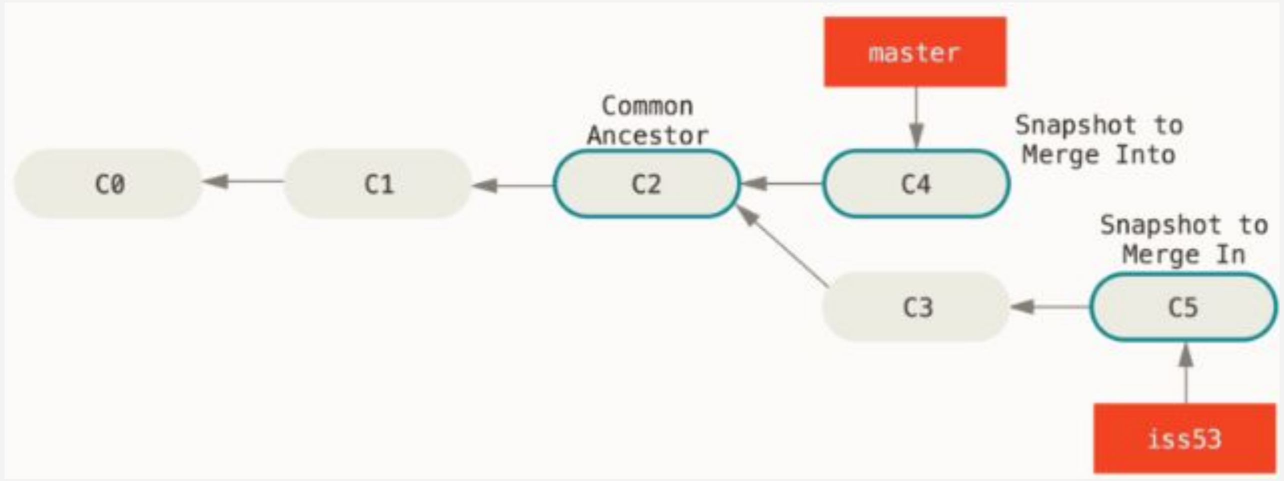
Make a new change, and the branches diverge

```
git commit
-a -m ‘other
changes’
```



Merging

→ To merge the snapshot from iss53:
  - `git checkout master`
  - `git merge iss53`


master
Common Ancestor
Snapshot to Merge Into
C0
C1
C2
C4
Snapshot to Merge In
C3
C5
iss53