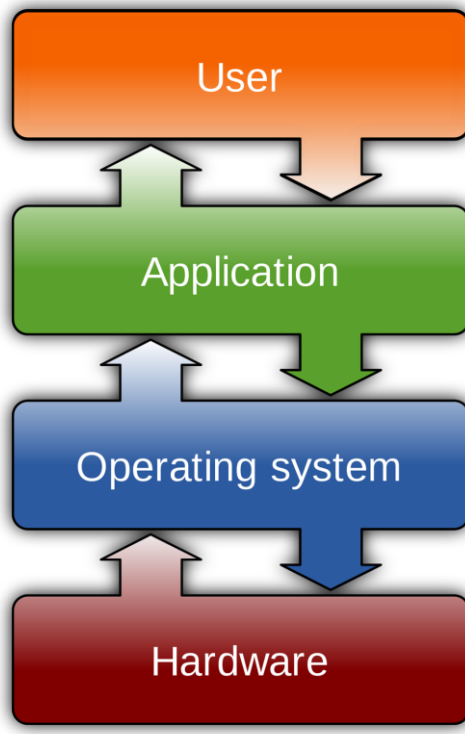
Assignment 1

What is an Operating System?

- Most important software that runs on a computer
- Responsible for many things
  - Managing other software
  - Abstracting hardware
- Brief history of Operating Systems: http://www.informit.com/articles/article.aspx?p=24972
- OS Examples: Windows (Windows 10, 8), macOS, FreeRTOS, Linux, Unix



Source : https://en.wikipedia.org/wiki/Operating_system

Multiuser and Multi-process Operating System

- Multi-User OS- Allow many users to access/work on a single system at the same time (as long as they have their own terminal)
- Multi-Process OS- Allows many processes, programs and applications to run simultaneously.
- Variants :
  - Single User Single Task
  - Single User Multi Task
  - Multi User OS

Debian GNU/Linux

- Clone of UNIX
- Linux is just a kernel.
- What is a kernel?
  - Core of any OS
  - Allocates time and memory to programs
  - Interfaces applications with the physical hardware
  - Facilitates communication between different processes (inter-process communication (IPC))
- Linux distributions make the Linux kernel a completely usable OS by adding various applications
- Linux distribution = GUI + GNU utilities (cp,mv,ls,etc) + installation and management tools + GNU compilers (c/c++) + Editors(vi/emacs) + ….
- Shell : Interface between the user and kernel

Files and Processes

- Everything is either a process or a file
- **Process**: an executing program identified by PID
- **File**: collection of data
  - A document
  - Text of program written in high-level language
  - Executable
  - Directory
  - Devices

The Basics: Dealing with Files

- **mv**: move/rename a file
- **cp**: copy a file
- **rm**: remove a file
  - r: remove directories and their contents recursively
- **mkdir**: make a directory
- **rmdir**: remove an empty directory
- **ls**: list contents of a directory
  - d: list only directories
  - a: list all files including hidden ones
  - l: show long listing including permission info
  - s: show size of each file, in blocks

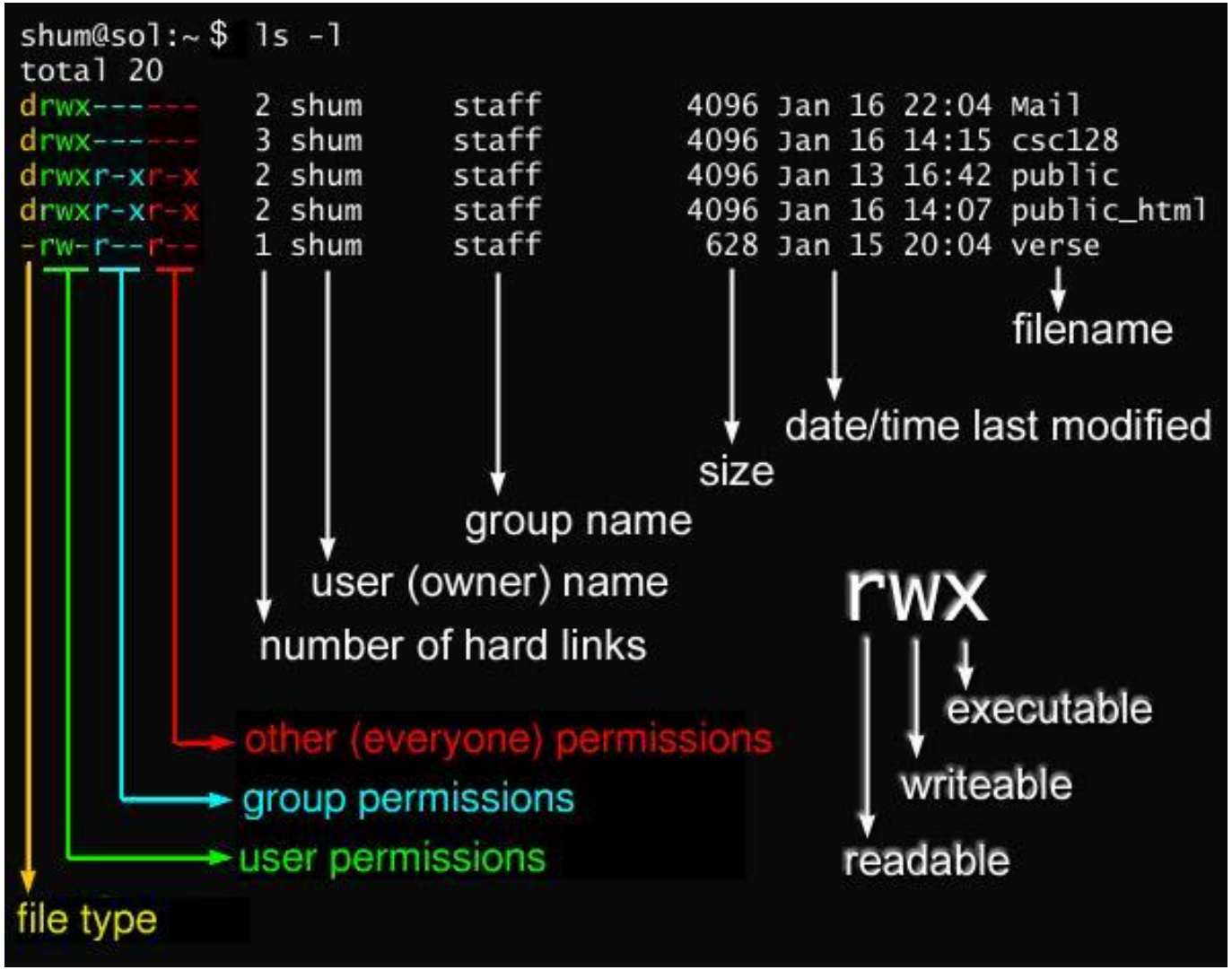
The Basics: Changing File Attributes

- **ln :** creates a link
  - **Hard links** : Point to physical Data
  - Additional name for an existing file
    - ln file1 hlink1
  - **Soft Links/ Symbolic Links (-s):** Point to file
    - ln –s <source file> <my file>
- **touch**: update access & modification time to current time
  - touch *filename*
  - touch -t 201101311759.30 *filename*
    - Change filename’s access & modification time to (year 2011 January day 31 time 17:59:30)

What on Earth are Hard and Soft Links?

- Hard Links
  - An additional name for a file (.txt, .py, .doc, etc.)
  - If “original” is removed, the “copy” will still work
    - Copy is not actually a separate copy
  - Can’t point to directories (folders)
- Soft Links (AKA symbolic links AKA symlinks)
  - Don’t point to the file, point to *something* in the file system
  - Can point to directories, or remote files
  - Deleting original file renders symbolic link unusable

The Basics: File Permissions



File Permissions

- chmod
  - read (r), write (w), executable (x)
  - User, group, others

Reference	Class	Description
u	user	the owner of the file
g	group	users who are members of the file's group
o	others	users who are not the owner of the file or members of the group
a	all	all three of the above, is the same as *ugo*

chmod contd…

- **Numeric**

#	Permission
7	full
6	read and write
5	read and execute
4	read only
3	write and execute
2	write only
1	execute only
0	none

- **Symbolic**

Operator	Description
+	adds the specified modes to the specified classes
-	removes the specified modes from the specified classes
=	the modes specified are to be made the exact modes for the specified classes

Mode	Name	Description
r	read	read a file or list a directory's contents
w	write	write to a file or directory
x	execute	execute a file or recurse a directory tree

Special permissions

- **setuid** : set user ID on execution
- Permits users to run certain programs with escalated privileges
- E.g. : chmod u+s file1
- When an executable file's setuid permission is set, users may access the program with a level of access that matches the owner
- E.g. passwd command

```
ls -l /usr/bin/passwd
```

```
-rwsr-xr-x 1 root 54192 Nov 20 17:03 /usr/bin/passwd
```

Special permissions contd...

- **setgid** : Grants permission of the group which owns the file
- E.g. : chmod g+s file2

```
ls -l myfile2
```

```
-rw-r-sr-- 1 user 0 Mar 6 10:46 myfile2
```

Basic Shell Commands

- man
- cat
- head
- tail
- du
- ps
- kill
- diff
- cmp
- wc
- sort
- echo

The Basics: Redirection

- **> *file*:** write stdout to a file
- **>> *file*:** append stdout to a file
- **< *file*:** use contents of a file as stdin

find command

- -type: type of a file (e.g: directory, symbolic link)
- -perm: permission of a file
- -name: name of a file
- -user: owner of a file
- -maxdepth: how many levels to search

find contd…

- ?: matches any single character in a filename
- *: matches one or more characters in a filename
- []*:* matches any one of the characters between the brackets. Use ‘-’ to separate a range of consecutive characters.
- Examples:
  - find . -name my*
  - find . -name my* -type f
  - find / -type f -name myfile

wh commands

- `whatis` <string>: returns Name section of man pages containing given string

```
~ tylerdavis$ whatis python
pydoc(1)          - the Python documentation tool
python(1)         - an interpreted, interactive, object-oriented programming language
pythonw(1)        - run python script allowing GUI
```

- `whereis` <command>: checks the standard binary directories for the specified programs, printing out the paths of any it finds

```
~ tylerdavis$ whereis python
/usr/bin/python
```

- `which` <command>: searches the path for each executable file that would be run had these commands actually been invoked

```
~ tylerdavis$ which python
/usr/local/bin/python
```

diff command

- A file comparison utility that outputs the differences between two files.
- Usage:
  - diff file1 file2
  - diff –u file1 file2 (unified format)

Text Editors: Vi

- Open a file- **vi <filename> or vim <filename>**
- Close a file- **:q**
- Save a file- **:w**
- Save and close a file- **:wq**
- Modes:
  - Normal: Enter commands
  - Insert: Insert text
  - Visual: Like normal, but you can highlight
  - Replace: Like insert, but you replace characters as you type
  - Recording: Record a sequence of key sequences

Text Editors: Emacs

- “The customizable, extensible, self documenting display editor”
- Customizable (no programming)
  - Users can customize font, colors, etc. in ~/.Emacs
- Extensible (programming required)
  - Run Lisp scripts to define new commands
- Self-documenting
  - C-h r (manual) and C-h t (tutorial)
- Real-time
  - Edits are displayed as you make them

Emacs: Editing Basics

- **Insert text** by simply typing it
- **Undo** by typing C-x u
- **Save changes** by typing C-x C-s
- **Copy, cut, paste**
  - C-space (starts selecting region)
  - M-w (copy a region)
  - C-w (cuts a region)
  - C-k (kill a line)
  - C-y (yank/paste)

Emacs: Buffers

- Buffer: Where the text you're editing lives
- Save current buffer to file
    - C-x C-s
- List all the buffers
    - C-x C-b
- Save current buffer with a specified file name
    - C-x C-w [filename]
- Add a new file to buffer
    - C-x C-f
- Visit *scratch* buffer
    - C-x b

Emacs: Navigating in a Buffer

Keystrokes	Action
C-p	Up one line
C-n	Down one line
C-f	Forward one character
C-b	Backward one character
C-a	Beginning of line
C-e	End of line
C-v	Down one page
M-v	Up one page
M-f	Forward one word
M-b	Backward one word
M-<	Beginning of buffer
M->	End of buffer
C-g	Quit current operation

Emacs: Bonus Commands

- Search – C-s
- Replace – M-%
- Accessing menu – F10
- Switch buffer – C-x b
- Switch current window – C-x o
- Kill the current window – C-x 0 (zero)
- Help – C-h

Other features

- Emacs as lisp interpreter
  - **M-x emacs-lisp-mode**
  - C-x C-e : Evaluate expression up to point
- Run shell commands
  - M-! <command>, **M-x shell** (interactive shell)
- Emacs as an IDE
  - **M-x compile**, then specify command to compile
  - Tip for homework: gcc hello.c –o hello
  - Run the executable by running the shell command
  - ./hello

Assignment 2

Character Sets and Encodings

- • ASCII (both a character set and an Encoding):
  - 128 Characters
- Encoded with bytes for each character
  - Byte values 128-255 not used (invalid)
  - Uniform Length Code
- • Unicode (Character set):
  - 1,112,064 valid code points
- • UTF-8 (Encoding):
  - All code points 1-4 bytes (var length)
  - Prefix-free code -- No 2 code points have same prefix
  - ASCII is a subset with same byte representation
- Other encodings exist as well
  - https://en.wikipedia.org/wiki/Comparison_of_Unicode_encodings

Example Character Set: ASCII

- Table shows all 128 characters + their encodings

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Environment Variables

Variables that can be accessed from any child process

Common ones:

- **HOME**: path to user's home directory
- **PATH**: list of directories to search in for command to execute

Change value:

export VARIABLE=…

Locale

A locale

- Set of parameters that define a user's cultural preferences
  - Language
  - Country
  - Other area-specific things

`locale` command

prints information about the current locale environment to standard output

Locale

- Where do these environment variables get set from?
  - Try man locale, see if it says where the information comes from
  - See if it says how to make your own custom locale
    - see if you can find where locale live on disk

Locale Settings Can Affect Program Behavior!!

Default sort order for the sort command depends:

- LC_COLLATE='C': sorting is in ASCII order
- LC_COLLATE='en_US': sorting is case insensitive except when the two strings are otherwise equal and one has an uppercase letter earlier than the other.

Other locales have other sort orders!

`sort`, `comm`, and `tr`

sort: sorts **lines** of **text** files

- Usage: sort [OPTION]…[FILE]…
- Sort order depends on locale
- C locale: ASCII sorting

comm: compare two **sorted** files **line by line**

- Usage: comm [OPTION]…FILE1 FILE2
- Comparison depends on locale

tr: translate **or** delete characters

- Usage: tr [OPTION]…SET1 [SET2]
- echo "12345" | tr "12" "ab”
- echo "password a1b2c3" | tr -d [:digit:]
- echo “abc” | tr [:lower:] [:upper:]

UNIX Wildcards : Link

- A *wildcard* is a character that can stand for all members of some class of characters
- The * wildcard
  - The character * is a wildcard and matches **zero or more character(s)** in a file (or directory) name. (ls list* or ls *list)
- The ? Wildcard
  - The character *?* will match **exactly one character**. (ls ?list OR ls list?)
- The [] Wildcard
  - A pair of [] represents **any of the characters enclosed** by them (ls *[0-9]*)

Regular Expressions (regex)

- A regex is a special text string for describing a certain search pattern
- Quantification
  - How many times of previous expression?
  - Most common quantifiers: ?(0 or 1), *(0 or more), +(1 or more)
- Alternation
  - Which choices?
  - Operators: [] and |
  - E.g Hello|World , [A B C]
- Anchors
  - Where?
  - Characters: ^(beginning) and $(end)

regex contd…

- ^ start of line
- $ end of line
- \ turn off special meaning of next character
- [] match any of enclosed characters, use – for range
- [^] match any characters except those enclosed in []
- . match a single character of any value
- * match 0 or more occurrences of preceding character/expression
- + match 1 or more occurrences of preceding character/expression

regex contd…

Expression	Matches
tolstoy	The seven letters tolstoy, anywhere on a line
^tolstoy	The seven letters tolstoy, at the beginning of a line
tolstoy$	The seven letters tolstoy, at the end of a line
^tolstoy$	A line containing exactly the seven letters tolstoy, and nothing else
[Tt]olstoy	Either the seven letters Tolstoy, or the seven letters tolstoy, anywhere on a line
tol.toy	The three letters tol, any character, and the three letters toy. Anywhere on a line
tol.*toy	The three letters tol, any sequence of zero or more characters, and the three letters toy. Anywhere on a line

regex contd.

- Parentheses allow you to apply quantifiers to sequences of characters
  - (abc)+ will match abc, abcabc, etc.
- Curly braces allow custom quantifiers
  - (abc){2} will only match abcabc
  - (abc){2,3} will match 2 or 3 repetitions
  - (abc){4,} will match 4 or more repetitions

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
- BRE tends to take things more literally
- In BRE ‘ ? ’, ‘ + ’, ‘ { ’, ‘ } ’, ‘ (’, and ‘) ’ lose their special meanings
- -E : Uses extended regular expressions

sed (Stream Editor)

- Modifies the input as specified by the command(s)
- Can be used for:
- Printing specific lines or address ranges
    - sed –n ‘1p’ sedFile.txt
        - prints first line
    - sed –n ‘1,5p’ sedFile.txt
        - prints 1st, 5th lines
    - sed –n ‘1~2p’ sedFile.txt
        - prints 1st line, then every second line after
- Deleting text
    - sed '1~2d' sedFile.txt
- Substituting text - s/regex/replacement/flags
    - sed 's/cat/dog/' file.txt
    - sed 's/cat/dog/g' file.txt
    - sed 's/<[^>]*>//g' a.html
    - `sed 's/`***`regExpr`***`/`***`replText`***`/'` filename

sed

- Substitution Flags
  - g: Replaces all matches
  - Number: replaces only the Numberth match **of every line**
- Regexes used with sed:
  - ^ $. * []
  - Use sed –r to evaluate ? Or +
    - specifies the Extended regex mode

sed contd..

- sed –n 12,18p file.txt
  - print lines 12 and 18
- sed 12,18d file.txt
  - delete lines 12 and 18
- sed ‘1~3d’ file.txt
  - delete first line, then every 3rd after
- sed ‘1,5 s/line/Line/g’ file.txt
  - on lines 1 and 5, replace every “line” with “Line”

sed contd.

- sed ‘/pattern/d’ file.txt
  - delete lines containing the pattern
- sed ‘/regexp/!d’ file.txt
  - delete lines not containing the pattern

grep

- A Unix command to search files/text for the occurrence of a string of characters that matches a specified pattern
- Usage:
  - grep [option(s)] pattern [file(s)]
- grep –r ‘potato’ .
  - Run grep on every file below the current directory, printing lines matching
- grep –c ‘line’ file.txt
  - Print number of matching lines
- grep –n ‘line’ file.txt
  - prefix output with line numbers

grep

- grep –E or egrep treats the expression as extended regular expression
- grep searches the contents of the files you pass in
- If you want to search for files, pass output of ls into grep

Useful grep Options

- -l
  - Rather than printing all matching lines, print filenames with matches
- -L
  - Opposite of -l, print all files without matches
- -v
  - Return only lines that do not match the given pattern

awk

- awk is more than a command; it’s a programming language by itself
- Utility/language for data extraction
- awk views a text file as records and fields
  - similar to rows and columns
- Usage:
  - awk ‘/search pattern/ {Actions}’ file
- Examples:
  - awk ‘{print;}’ file.txt // print the file line by line; default behaviour
  - awk ‘/Hello/ {print;}’ file.txt // prints lines which matches Hello
  - awk ‘{print $1,$2;}’ file.txt // prints only specific fields
  - awk –F’Hello’ ‘{print $2}’ // prints second column in between the occurrences of the specified pattern

Piping and Redirection

- Every program we run on the command line automatically has three data streams connected to it.
  - STDIN (0) - Standard input (data fed into the program)
  - STDOUT (1) - Standard output (data printed by the program, defaults to the terminal)
  - STDERR (2) - Standard error (for error messages, also defaults to the terminal)

Piping and redirection is the means by which we may connect these streams between programs and files to direct data in useful ways.

More About Piping and Redirection

- Basic I/O Redirection
  - Most programs read from stdin
  - Write to stdout
  - Send error messages to stderr

  - Task: Piping and Redirection
  - Create a file test.txt with numbers 1-5 in descending order in each line
    - Delete all the new line characters (with tr command and redirection) and redirect output to test1.txt
    - Now, first sort the file and then repeat the above step; but instead of redirection, now append the output to test1.txt

Assignment 3

Compiled vs Interpreted Languages

- Compiled Languages
  - Examples?
    - C,C++,Java
  - First Compiled
  - Source code to object code; then executed
  - Run faster (generally)
  - Applications:
    - Typically run inside a parent program like scripts, can be compiled and used on any platform (eg. Java)

- Interpreted Languages
  - Examples?
    - Python, JavaScript, Shell Scripting
  - No compilation required. Directly interpreted!
  - Interpreter reads program, translates into internal form and executes
  - Runs slower than a high level lang
  - Applications:
    - Automation, extracting information from a data set, but anything you want

Self-Contained Scripts: The #! First Line

- Shell hands program to kernel, asks for it to be run
  - Knows how to do this for compiled programs
  - Won't work for scripts
    - Need to make script executable!
      - chmod +x ./my_script
- Shebang at top of script specifies the interpreter to use
  - #!/bin/bash
  - #!/bin/awk –f
  - #!/bin/sh
  - #!/usr/bin/env python3
    - use the interpreter "python3" using the first python3 in the PATH
- Shell may select a default (python 2, /bin/sh)

Variables in Shell Scripts

- Start with a letter or underscore and may contain any number of following letters, digits, or underscores
- Hold string variables

- $ myvar=this_is_a_long_string_that_does_not_mean_much //Assign value
- $ echo $myvar //Print the value

this_is_a_long_string_that_does_not_mean_much

- first=firstName middle=s last=lastName Multiple assignments allowed on one line
- fullname="$first $middle $last" Double quotes required here, for concatenating
- fullname=“abc xyz mno" Use quotes for whitespace in value
- oldname=$fullname Quotes not needed to preserve spaces in value

Variables in shell script contd…

- **Escape Character ** - Literal value of following character
  - echo \|
- **Single Quote** - Literal Meaning of all within ‘ ’
  - $hello=1
  - $str=‘$hello’
  - echo $str -> $hello
  - echo “howdy!” # this will fail as it tries to interpret “!”
  - echo ‘howdy!’ # this succeeds
- **Double Quote** - Literal meaning except for $, ` and \.
  - $hello=1
  - $str=“abc$hello”
  - $word=”abc`pwd`”
  - echo $str -> abc1
  - echo $word -> abc/u/cs/grad/joe

Variables in shell scripts contd…

- Special Variables: certain characters reserved as special variables
    - $: PID of current shell
    - #: number of arguments the script was invoked with
    - n: nth argument to the script
    - ?: exit status of the last command executed
    - echo $$; echo $#; echo $2; echo $?;
- scalar variable vs array variable:
    - array_name[index]=value; echo ${array_name[index]}

Loops

- for var in list_values

```
do
    command 1
    ..
    command n
done
```

- while condition

```
do
    command 1
    ..
    command n
done
```

```
ALL=`ls -a $dir | sort`
declare -a ARRAY
count=0
for FILE in $ALL
        do
        ARRAY[$count]=$FILE
        .....
        done
```

```
for i in "${ARRAY[@]}”
        do
        …
        done
```

Conditional and Unconditional Statements

- **Conditional**
  - if…then…fi
  - if…then…else…fi
  - if…then…elif..then…fi
  - case…esac
  - Do `man test` to see what else you can check
- **Unconditional**
  - break
    - goes to first line after loop
  - continue
    - jumps to start of loop for new iteration

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

Conditionals cont.

- There is also a [[expression]] check format
- Has some more features
- Not supported everywhere
  - See the following stack overflow post
  - See also http://mywiki.wooledge.org/BashFAQ/031

Math/Expressions

- Math needs to be written as $((Expression))
  - Equivalent to $(expr Expression)
    - in other words, call the command expr with my Expression
- Look at man expr for more
  - Look at the authors list, see any familiar names?

```
#!/usr/bin/sh

COUNT=0

while [ $COUNT -lt 10 ]
  do
    COUNT=$(($COUNT + 1))
    echo $COUNT
  done
```

Functions

- Group commands for later execution
- Can invoke like a normal command
- Access positional arguments via $1 … $k
- Can return your own exit status with `return code`
  - other output through stdout / piping
- More available in manual and various tutorials

```
#!/bin/sh

Hello () {
  local greeting=”Yo”
  echo “${greeting} $1 $2”
  return 0
}

Hello abc def
```

Parameter Expansion

- $parameter replaces the symbol parameter with a value
  - this is parameter expansion
- Does much more than just retrieving the string a variable corresponds to
- Can even do replacements like sed!
  - ${parameter/pattern/string}
    - replaces first instance of pattern in expanded parameter with string
- See the manual for much more
  - note that this is from the bash manual

Bonus Shell Tidbits

- @ - Arguments to script (echo $@)
- ${parameter} - like $parameter, but less ambiguous
- $(command) - Command is executed in a subshell (at least in Bash)
  - curly braces do NOT launch a subshell
  - subshells can execute in parallel
- command & - run the command in the background
- ; - Separator between commands
  - separator for multiple commands on one line
  - Used for alternate if/while syntax
- command1 && command2 - returns 0 as exit status iff command1 and command2 both return 0 exit status. Won't run command2 if command1 fails

Bonus Shell Tidbits

- command1 || command2 - returns logical or of exit statuses, similar to && in execution
- set -u - Error when attempting to access unset variables
- set -e - Exit immediately if anything at all returns a nonzero exit status
  - Be careful of this one, it can bite you
  - can work around with failing_command || echo “hello”
    - OR makes the overall return value 0
- Avoid parsing ls in your scripts
  - ls >= 8.25 correctly escapes when printing to terminal, but not guaranteed you’d have that
  - Easier to loop over files in current directory with a for loop (see above link)

Helpful Commands

- basename
  - basename /usr/bin/emacs -> emacs
  - man basename
  - basename include/stdio.h .h -> stdio

Task

- Create a file with following text (singers.txt):
  1, Justin Timberlake
  2, Taylor Swift
  3, Mick Jagger
  4, Lady Gaga
  5, Johnny Trash
  6, Elvis Presley
  7, John Lennon
  - Print all singers having first name as ‘John’
    - Ans: sed -n '/John /p’ singers.txt > john.txt
  - Change all , to) in the file
    - Ans: sed 's/,/)/' singers.txt
  - Append a period to each line
    - Ans: sed 's/$/./' singers.txt

Task

- Create a file f1.txt with the following content

  UPPER CASE LINE

  Lower case line

  Break;

  empty LINE

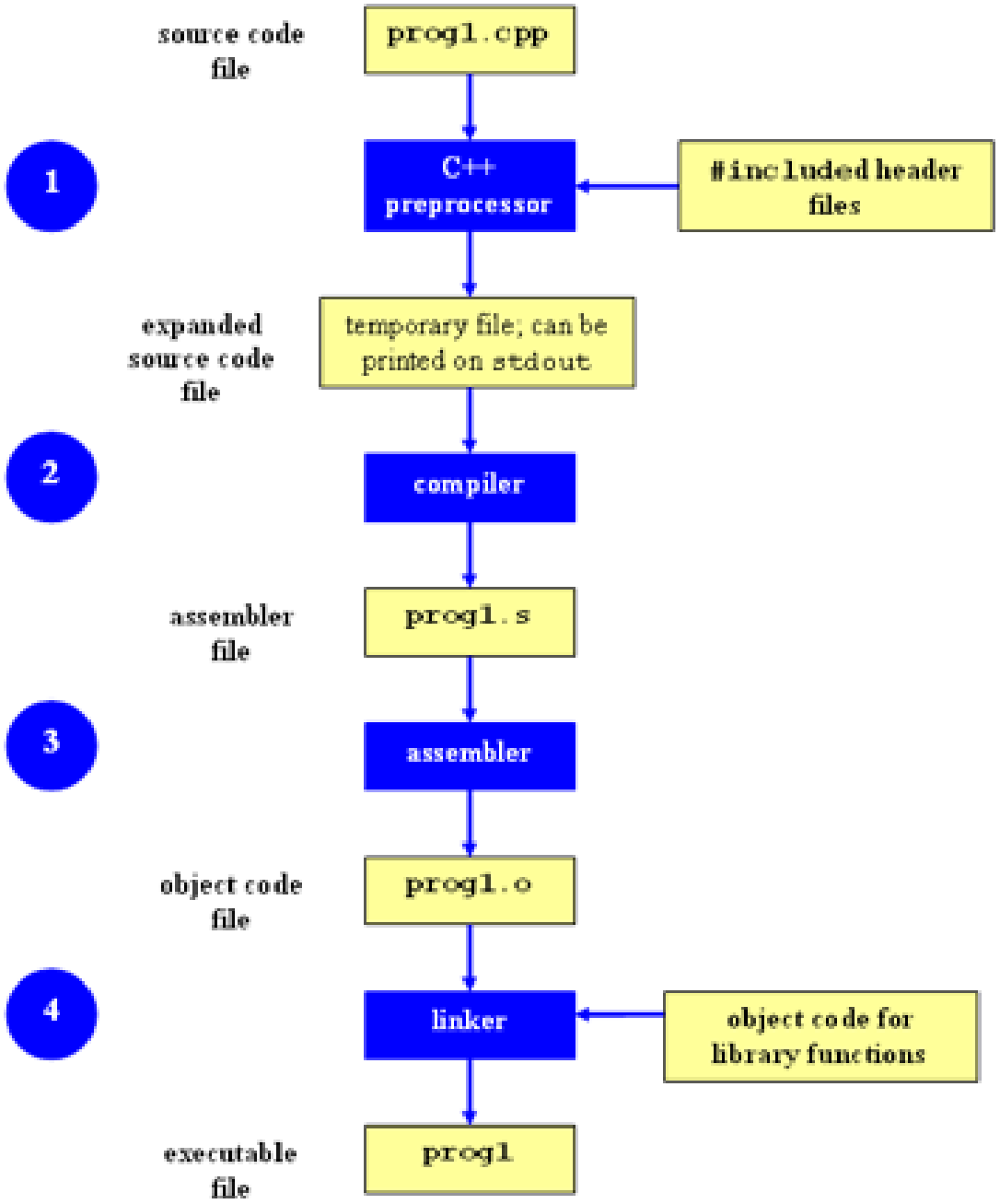
  Last line

- Check for the given string ‘line/LINE’ in *text* files which *start with ‘f’* and *end with a number.txt*
- Replace all upper case letters to lower case using tr
  - Ans: tr ‘[:upper:]’ ‘[:lower:]’ < f*[1-9].txt
- Repeat the same using sed
  - Ans: sed 's/.*/\L&/g' f*[1-9].txt

Task

- Write a regex to validate an email with the following rules
  - It should start with a lower case letter only
  - It should have the domain name as gmail.com (string representation is @gmail.com)
  - Ans: ^[a-z].*@gmail\.com

Compilation Process



Compilation Example

- shop.cpp
  - #includes shoppingList.h and item.h
- shoppingList.cpp
  - #includes shoppingList.h
- item.cpp
  - #includes item.h
- How to compile?
  - **g++ -Wall shoppingList.cpp item.cpp shop.cpp –o shop**

What if…

- **We change one of the header or source files?**
  - Rerun command to generate new executable
- **We only made a small change to item.cpp?**
  - not efficient to recompile shoppinglist.cpp and shop.cpp
  - Solution: avoid waste by producing a separate object code file for each source file
    - g++ -Wall –c item.cpp… (for each source file)
    - g++ item.o shoppingList.o shop.o –o shop (combine)
    - Less work for compiler, saves time but more commands

What if…

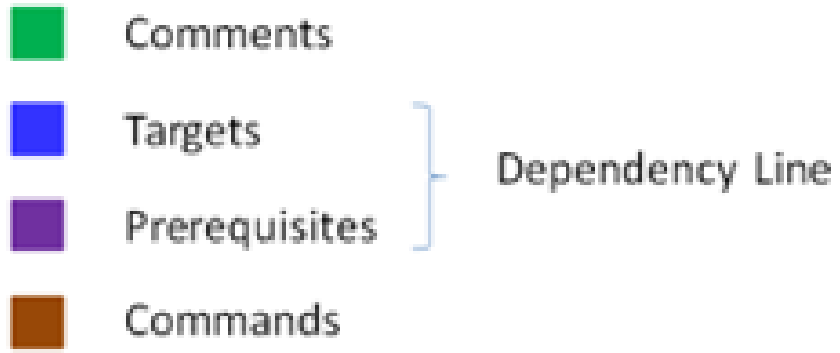
- We change item.h?
  - Need to recompile every source file that includes it
  - Need to recompile every source file including a header that includes it
    - This means item.cpp and shop.cpp
- Hard to keep track of dependencies ourselves

Solution: Make

- Utility for managing software projects
- Compiles files, ensures they are up to date
- Efficient compilation
  - Only recompiles things that need to be recompiled
- Gets information from file called a “makefile”
- Manual describes makefiles and much more

Example Makefile

```
# Makefile - A Basic Example
all : shop  #usually first
shop : item.o shoppingList.o shop.o
  g++ -g -Wall -o shop item.o shoppingList.o shop.o
item.o : item.cpp item.h
  g++ -g -Wall -c item.cpp
shoppingList.o : shoppingList.cpp shoppingList.h
  g++ -g -Wall -c shoppingList.cpp
shop.o : shop.cpp item.h shoppingList.h
  g++ -g -Wall -c shop.cpp
clean :
  rm -f item.o shoppingList.o shop.o shop
```



Build Process

- **Configure**
  - Script that checks details about the machine before installation
    - Dependencies between packages
  - Creates ‘Makefile’
- **Make**
  - Requires ‘Makefile’ to run
  - Compiles program and creates executables
- **make install**
  - make utility searches for a label named install within the Makefile, and executes only that section of it
  - Copies executables into the final directories (system directories like /usr/bin)

Python Conditional statements

```
x = 5
y = 8
if x == 7:
    print(“The value of X was {}”.format(x))

if x <= 7:
    print(“hello”)
elif y <= 7:
    print (“x fails, y good”)
else:
    print(“Not good”)

if (x <=7) and (y == 8):
   print(“Condition valid”)
```

Python Lists

- Like arrays, but not
  - Can change their size
  - Can hold items of different types

```
x = [1, 1.5, “Goodbye”]

y = []

z = x + x

x[3] # Error!

x[-1]  # “Goodbye”

x.append(y)

x.append(x)

x[0] # 1
```

Slicing

- Allows for creative access into lists + strings
- Format: my_list[start:stop:step]
- Alternate format: my_list[start:stop]

x = “hello string”

x[1:] # “ello string”

x[:-1] # “hello strin”

x[0::2] # “hlosrn”

x[::-1] # “gnirts olleh”

Python For/While loops

- An iterable is anything you can iterate over
  - Can call next(X) on it

```
for x in iterable:

    print(x)
```

Dictionaries

- Map keys to values
  - Strings -> lists
  - Ints -> floats
  - Etc.
- Keys must be unique
- Support hete data types

```
X = {} # Empty dict

Y = {1: “hi”, “I”: 3}

X[“bye”] = 18

Y[1] # “hi”
```

Opening Files

```
with open(filename) as my_file:
    for line in my_file:
        print(line)
```

Exceptions

- A way to report type of problem + what went wrong
- Allows new code paths when errors are encountered
- Try/Except let you handle them

```
try:
    x = int(input("Please enter a number: "))
    break
except ValueError as e:
   print("Oops!  That wasn't a number!")
```

```
if True:
  raise ValueError("uh oh!"
```

Classes

- Let you keep state
- Let you reuse code
- Help you organize code

```
class Rectangle:
  def __init__(self, x, y):
        self.l = x
        self.b = y

  def getArea(self):
      return self.l * self.b
  def getPerimeter(self):
      return 2 * (self.l + self.b)
def main():
  rect = Rectangle(3, 4)
  print("Area of Rectangle:", rect.getArea())
  print("Perimeter:", rect.getPerimeter())
main()
```

```
#!/usr/bin/python
```
Tells the shell which interpreter to use

```
import random, sys
from optparse import OptionParser
```
Import statements, similar to include statements

Import OptionParser class from optparse module

```
class randline:
    def __init__(self, filename):
        f = open (filename, 'r')
        self.lines = f.readlines()
        f.close ()

    def chooseline(self):
        return random.choice(self.lines)
```
The beginning of the class statement: randline

The constructor

Create a file handle

Read the file into a list of strings called lines

Close the file

The beginning of a function belonging to randline

Randomly select a number between 0 and the size of lines and return the line corresponding to the randomly selected number

```
def main():
    version_msg = "%prog 2.0"
    usage_msg = """%prog [OPTION]...
FILE Output randomly selected lines
from FILE."""
```
The beginning of main function

version message

usage message

```
parser = OptionParser(version=version_msg,
                      usage=usage_msg)
parser.add_option("-n", "--numlines",
            action="store", dest="numlines",
            default=1, help="output NUMLINES
            lines (default 1)")

options, args = parser.parse_args(sys.argv[1:])

try:
    numlines = int(options.numlines)
except:
    parser.error("invalid NUMLINES: {0}".
                      format(options.numlines))
if numlines < 0:
    parser.error("negative count: {0}".
                 format(numlines))
if len(args) != 1:
    parser.error("wrong number of operands")
input_file = args[0]
try:
    generator = randline(input_file)
    for index in range(numlines):
        sys.stdout.write(generator.chooseline())
except IOError as (errno, strerror):
    parser.error("I/O error({0}): {1}".
format(errno, strerror))

if __name__ == "__main__":
    main()
```

Creates OptionParser instance

Start defining options, action "store" tells optparse to take next argument and store to the right destination which is "numlines". Set the default value of "numlines" to 1 and help message.

options: an object containing all option args

args: list of positional args leftover after parsing options

Try block

get numline from options and convert to integer

Exception handling

error message if numlines is not integer type, replace {0} w/ input

If numlines is negative

error message

If length of args is not 1 (no file name or more than one file name)

error message

Assign the first and only argument to variable input_file

Try block

instantiate randline object with parameter input_file

for loop, iterate from 0 to numlines – 1

print the randomly chosen line

Exception handling

error message in the format of "I/O error (errno):strerror

In order to make the Python file a standalone program

Basic Data Types

- **int**
  - Holds integer numbers
  - Usually 4 bytes
- **float**
  - Holds floating point numbers
  - Usually 4 bytes
- **double**
  - Holds higher-precision floating point numbers
  - Usually 8 bytes (double the size of a float)
- **char**
  - Holds a byte of data, a character
- **void**

Pretty much like C++ basic data types, but NO **bool** before C99

size_t

- Unsigned integer

Pointers

- Variables that store memory addresses

Declaration

- <variable_type> *<name>;
  - int *ptr; //declare ptr as a pointer to int
  - int var = 77; // define an int variable
  - ptr = &var; // let ptr point to the variable var
- & symbol, e.g. &my_integer
  - Gets you the address where that value lives in memory

Dereferencing Pointers

- Accessing the value that the pointer points to

- Example:
  - double x, *ptr;
  - ptr = **&**x; // let ptr point to x
  - *****ptr = 7.8; // assign the value 7.8 to x

qsort Example

void qsort (void* base, size_t num, size_t size, int (*compar)(const void*,const void*));

Return value meaning for comparator function:

< 0	The element pointed by p1 goes before the element pointed by p2
= 0	The element pointed by p1 is equivalent to the element pointed by p2
> 0	The element pointed by p1 goes after the element pointed by p2

```
#include <stdio.h>
#include <stdlib.h>
int compare (const void * a, const void * b){
        return ( *(int*)a - *(int*)b );
}
int main () {
        int values[] = { 40, 10, 100, 90, 20, 25 };
        qsort (values, 6, sizeof(int), compare);
        int n;
        for (n = 0; n < 6; n++)
                printf ("%d ",values[n]);
        return 0;
}
```

Arrays

```
int my_numbers[5] = {1, 2, 3, 4, 5];

int* my_number_pointers[5];
```

Structs

- No classes in C
- Used to package related data (variables of different types) together
- Single name is convenient

```
struct Student {
        char name[64];
        char UID[10];
        int age;
        int year;
};
struct Student s;
```

```
typedef struct {
      char name[64];
      char UID[10];
      int age;
      int year;
} Student;
Student s;
```

C structs vs. C++ classes

- C structs cannot have member functions
- There's no such thing as access specifiers in C
- C structs don't have constructors defined for them

- C++ classes can have member functions
- C++ class members have access specifiers and are **private** by default
- C++ classes must have at least a default constructor

Dynamic Memory

- Memory that is allocated at runtime
- Allocated on the heap
- You MUST free it yourself!

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
- Keeps the content

void free (void *ptr);

- Frees the block of memory pointed to by *ptr*

Reading/Writing Characters

- **int getchar();**
  - Returns the next character from stdin
- **int putchar(int character);**
  - Writes a character to the current position in stdout

Formatted I/O

- int fprintf(FILE * fp, const char * format, …);
- int fscanf(FILE * fp, const char * format, …);
  - FILE *fp can be either:
    - A file pointer
    - stdin, stdout, or stderr
  - The format string
    - int score = 120; char player[] = “John”;
    - fp = fopen(“file.txt”, “w+”)
    - fprintf(fp, **“%s has %d points.\n”, player, score**);

Homework 4

- Write a C program called *sfrob*
  - Reads stdin byte-by-byte **(getchar)**
    - Consists of records that are newline-delimited
    - Each byte is frobnicated (XOR with dec 42)
  - Sort records without decoding (**qsort, frobcmp**)
  - Output result in frobnicated encoding to stdout **(putchar)**
  - Error checking (**fprintf**)
  - Dynamic memory allocation (**malloc, realloc, free**)

Using GDB

1. **Compile Program**
   - Normally: $ gcc [flags] <source files> -o <output file>
   - Debugging: $ gcc [other flags] –g <source files> -o <output file>
     - enables built-in debugging support
     - pass -O0 to turn off optimizations, which can make debugging difficult

2. **Specify Program to Debug**
   - $ gdb <executable>

     or

   - $ gdb
   - (gdb) file <executable>

Using GDB

3. **Run Program**
   - `(gdb) run` or
   - `(gdb) run [arguments]`
4. **In GDB Interactive Shell**
   - Tab to Autocomplete, up-down arrows to recall history
   - `help [command]` to get more info about a command
5. **Exit the gdb Debugger**
   - `(gdb) quit`

Runtime Errors

- Segmentation faults
  - Normally just says “segmentation fault”
  - If run in a debugger, we instead get the following, which lets us see exactly where and why we crashed

```
#include <stdio.h>

int main() {
   int my_val = 0;
   for (int i = 0; i < 100; i++) {
      my_val += i;
   }
   int secret_value = my_val;
   int *bro;
   *bro = 208;  // This is very bad!
   printf("My value: %d\n", secret_value);
   return 0;
}
```

```
* thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=1, address=0x0)
    frame #0: 0x0000000100000f48 hello`main at hello.c:10:10
   7               }
   8               int secret_value = my_val;
   9               int *bro;
-> 10              *bro = 208;
   11              printf("This is just a message: %d\n", secret_value);
   12              return 0;
   13          }
Target 0: (hello) stopped.
```

Breakpoints

- Breakpoints
  - used to stop the running program at a specific point
  - If the program reaches that location when running, it will pause and prompt you for another command
  - You can set as many as you want
- Example:
  - (gdb) break file1.c:6
    - Program will pause when it reaches line 6 of file1.c
  - (gdb) break my_function
    - Program will pause at the first line of my_function every time it is called
  - (gdb) break [*position*] if *expression*
    - Program will pause at specified position only when the expression evaluates to true
    - expression can be anything valid in the language you're debugging

Managing Breakpoints

- (gdb) delete [bp_number | range]
  - Deletes the specified breakpoint or range of breakpoints
- (gdb) disable [*bp_number* | *range*]
  - Temporarily deactivates a breakpoint or a range of breakpoints
- (gdb) enable [*bp_number* | *range*]
  - Restores disabled breakpoints
- If no arguments are provided to the above commands, all breakpoints are affected!!
- (gdb) ignore *bp_number iterations*
  - Instructs GDB to pass over a breakpoint without stopping a certain number of times.
    - bp_number: the number of a breakpoint
    - Iterations: the number of times you want it to be passed over

Displaying Data

- You've paused execution with a breakpoint, now you want to look inside the program
  - (gdb) print [/format] *expression*
    - prints the value of the specified expression in the specified format
    - Formats:
      - d: Decimal notation (default format for integers)
      - x: Hexadecimal
      - o: Octal
      - t: binary

Continuing Execution

- Four ways to continue execution after a breakpoint
  - **c or continue**: debugger will continue executing until next breakpoint
  - **s or step**: debugger will continue to next source line
  - **n or next**: debugger will continue to next source line in the current (innermost) stack frame
  - **f or finish**: debugger will resume execution until the current function returns. Execution stops immediately after the program flow returns to the function's caller
    - the function's return value and the line containing the next statement are displayed

Watchpoints

- Watch/observe changes to variables
  - (gdb) watch my_var
    - sets a watchpoint on my_var
    - the debugger will stop the program whenever the value of my_var changes
    - prints out new and old values
  - (gdb) rwatch expression
    - Execution stops whenever the expression is read

Process Memory Layout

- Processes think they own all the memory
  - Done via “Virtual Memory”
  - Some systems, such as microcontrollers, might not use this
- Exact calling conventions (registers vs stack, etc.) can vary across architecture and OS
- Blog post explaining this in x86 (non 64 bit)

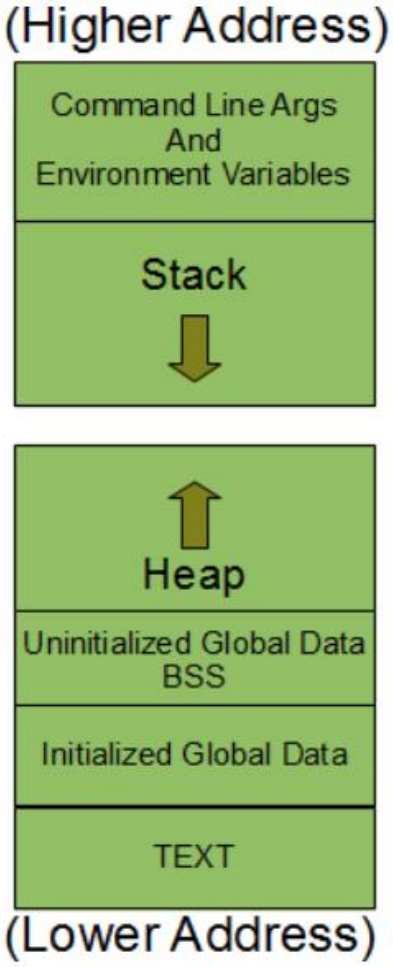


Image source : thegeekstuff.com

- TEXT segment
  - Contains machine instructions to be executed
- Global Variables
  - Initialized
  - Uninitialized
- Heap segment
  - Dynamic memory allocation
  - malloc, free
- Stack segment
  - Push frame: Function invoked
  - Pop frame: Function returned
  - Stores
    - Local variables
    - Return address, registers, etc
- Command Line arguments and Environment Variables

Process Memory Layout

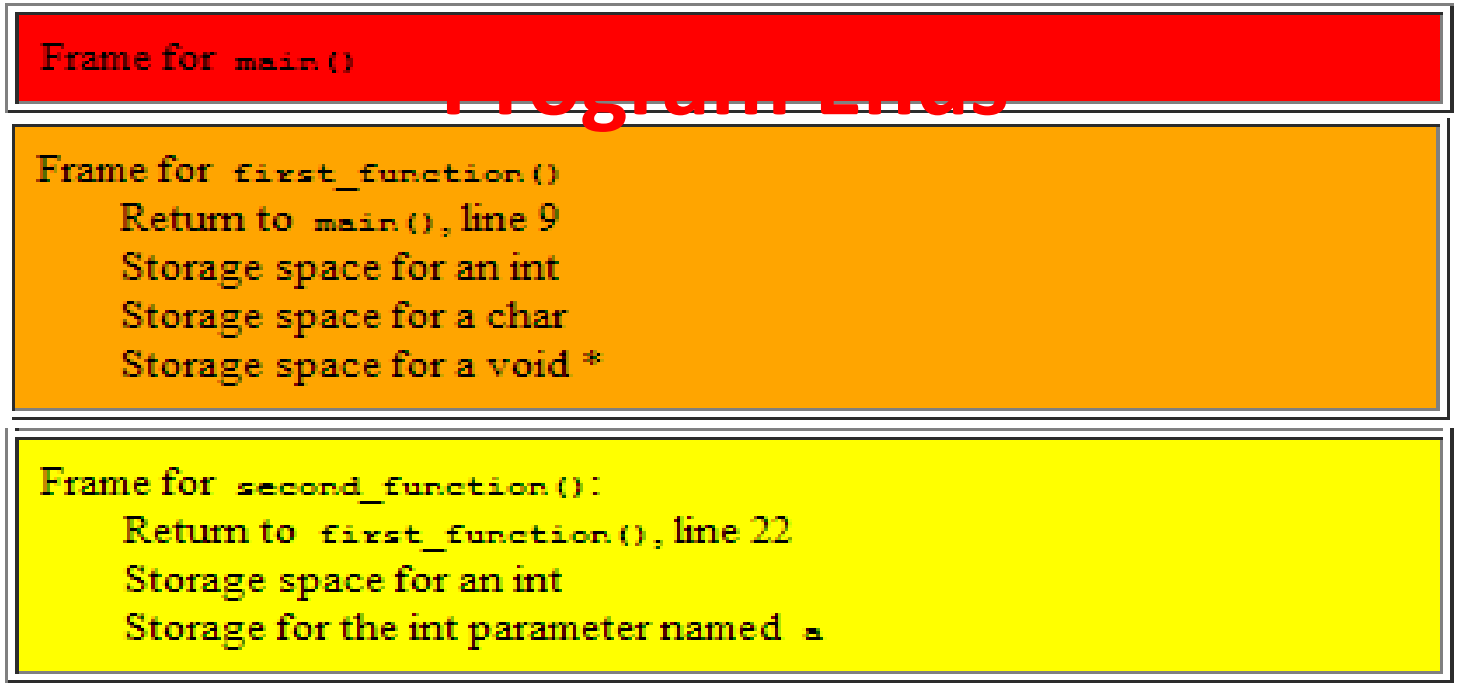
- There’s a ton of abstraction going on
- Don’t worry too much about what’s physically happening, that’s for another class
  - Caching
    - Write back / write through
    - Cache policies
    - Cache coherence on multicore / multichip machines
  - Virtual Memory
  - Paging

Stack Frames

- Every time a function is called, some memory is set aside for it
  - This memory is called a **stack frame**
- Stack frames consist of the following
  - Storage space for all local variables
  - Address of next line of code to execute after the function
  - The arguments to the function

Stack Frames and the Stack

```
1   #include <stdio.h>
2   void first_function(void);
3   void second_function(int);
4
5   int main(void)
6   {
7      printf("hello world\n");
8      first_function();
9      printf("goodbye goodbye\n");
10
11     return 0;
12  }
13
14
15  void first_function(void)
16  {
17     int imidate = 3;
18     char broiled = 'c';
19     void *where_prohibited = NULL;
20
21     second_function(imidate);
22     imidate = 10;
23  }
24
25
26  void second_function(int a)
27  {
28     int b = a;
29  }
```



Call to second_function() is made, unused stack memory is used to create a stack frame for second_function(). The frame holds storage space for an int and the current address of execution within second_function()

Analyzing the Stack in GDB

- (gdb) backtrace|bt
  - Shows the call trace (the call stack)
  - Without function calls:
    - #0 main () at program.c:10
    - one frame on the stack, numbered 0, and it belongs to main()
  - After call to function display()
    - #0 display (z=5, zptr=0xbffffb34) at program.c:15
      #1 0x08048455 in main () at program.c:10
    - Two stack frames: frame 1 belonging to main() and frame 0 belonging to display().
    - Each frame listing gives
      - the arguments to that function
      - the line number that's currently being executed within that frame

Analyzing the Stack

- (gdb) info frame
  - Displays information about the current stack frame, including its return address and saved register values
- (gdb) info locals
  - Lists the local variables of the function corresponding to the stack frame, with their current values
- (gdb) info args
  - List the argument values of the corresponding function call

Other Useful Commands

- (gdb) info functions
  - Lists all functions in the program
- (gdb) list
  - Lists source code lines around the current line

Kernel

- The kernel is the core of the OS
  - It’s a program too!
  - Interface between hardware and software
  - Controls access to system resources: memory, I/O, CPU
  - Manages CPU resources, memory resources, processes
  - Lowest layer above the CPU
  - Ensure protection and fair allocations

Which Code is Trusted?
The Kernel *ONLY*

- Core of OS software **executing in supervisor** state
- **Trusted software:**
  - Manages hardware resources (CPU, Memory and I/O)
  - Implements protection mechanisms that could not be changed through actions of untrusted software in user space

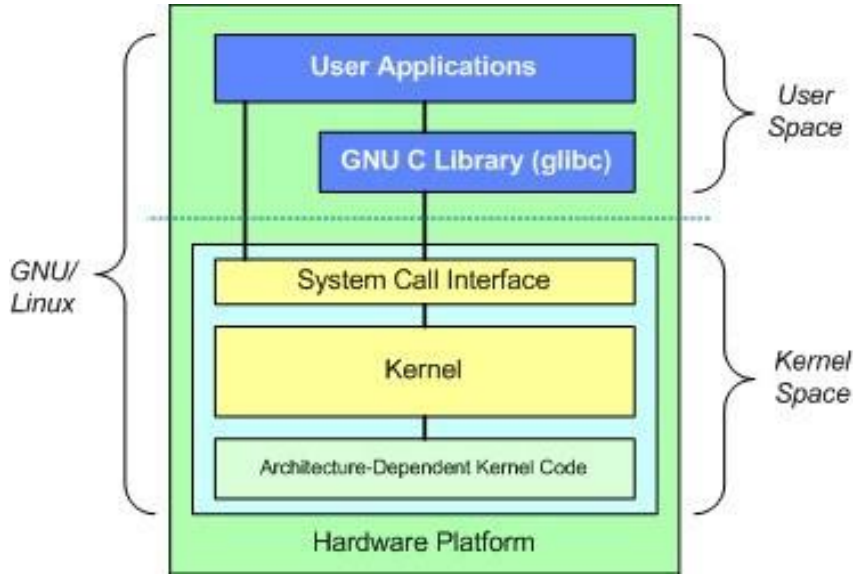
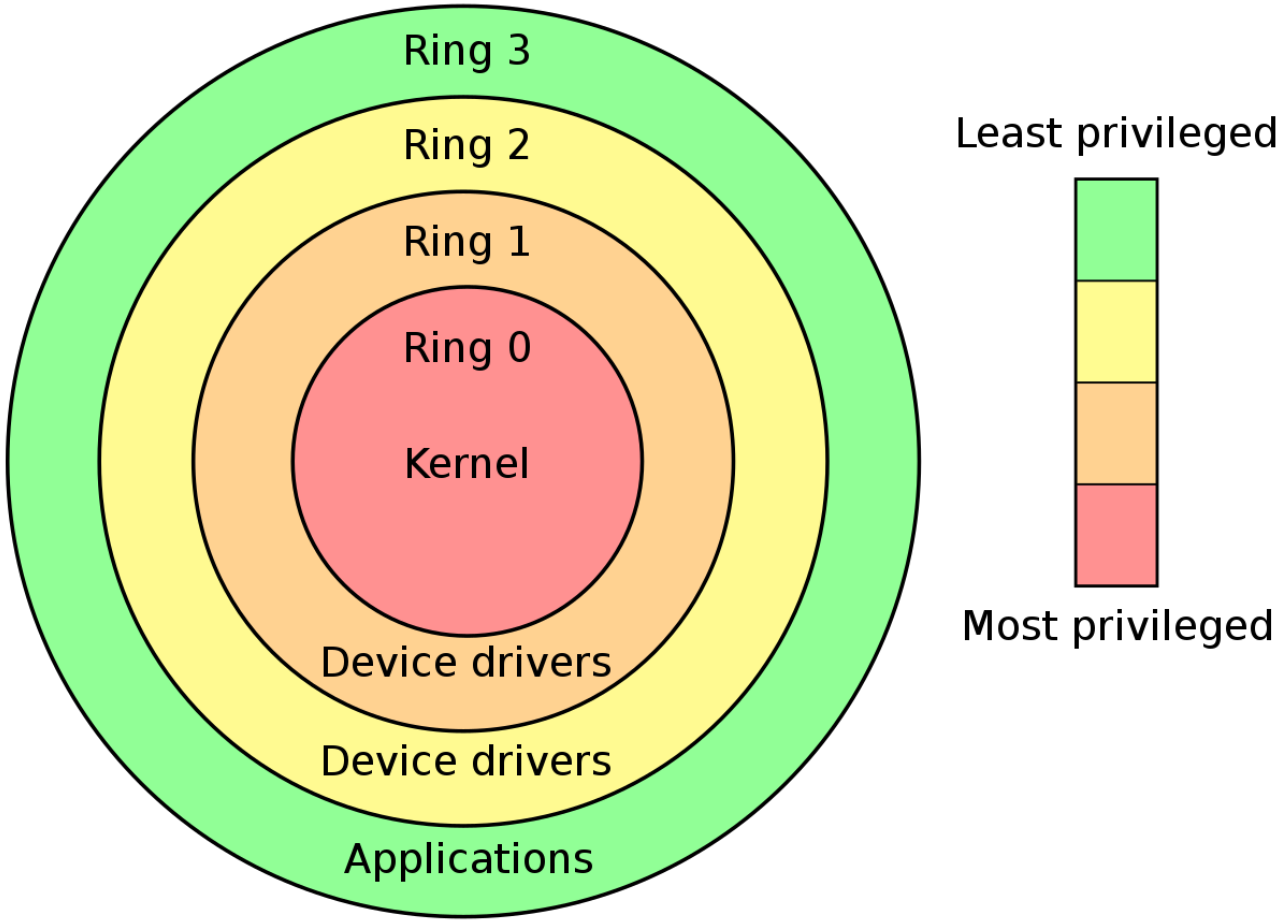


Image by: Tim Jones (IBM)

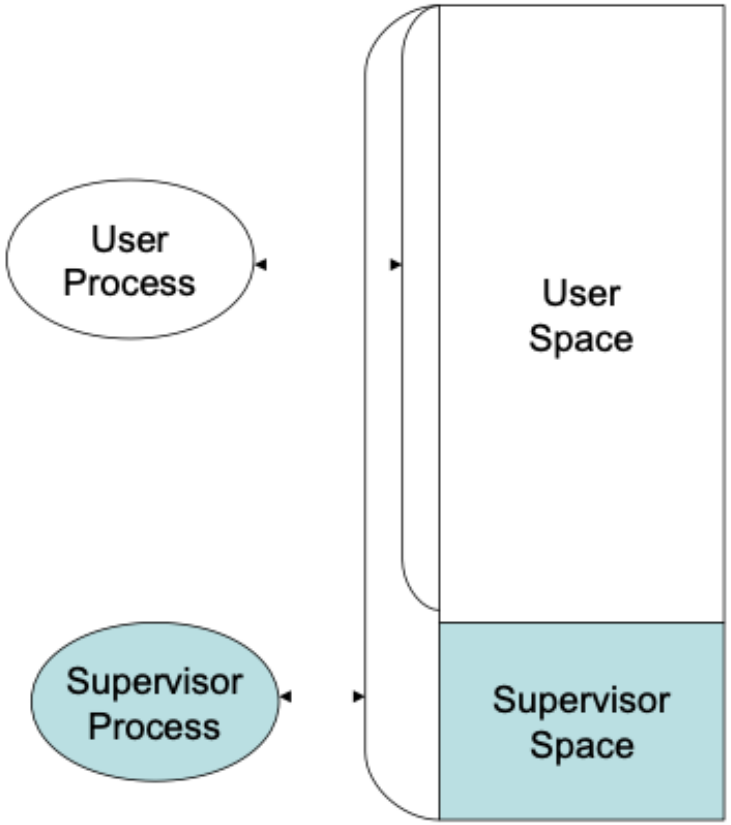
CPU Modes

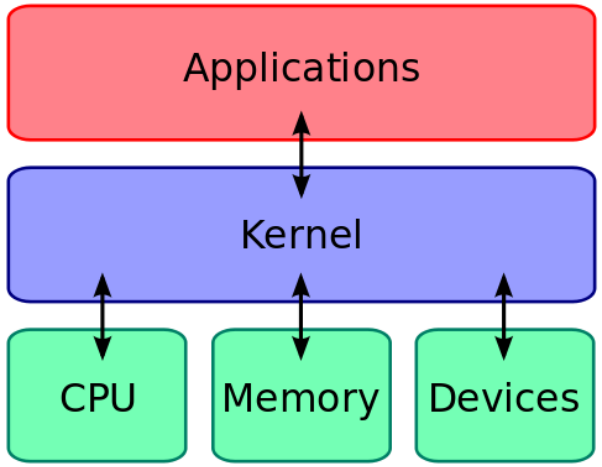
- Hardware contains a mode bit
  - e.g. 0 for kernel mode, 1 for user mode
  - Varies by architecture
    - x86 has 4 privilege levels
    - We'll stick to two
- User mode
  - Certain instructions unavailable
  - Can only interact with subset of memory
- Kernel mode
  - All instructions available
  - Can interact with any memory it wants



User space vs Kernel Space

- Virtual memory means that memory addresses are faked
- Memory split into two areas
  - User space
  - Kernel space
- User space
  - Where user programs live + run
  - Keeps user programs “in the dark”
- Kernel space
  - Where kernel runs
  - Holds code for kernel
  - We only trust this kernel code!





System Calls

- Special type of function that:
  - Provide interface between user programs and OS
  - Used by user-level processes to request a service from the kernel
  - Changes the CPU's mode from user mode to kernel mode to enable more capabilities
  - Is part of the kernel of the OS
  - Verifies that the user should be allowed to do the requested action and then does the action (kernel performs the operation on behalf of the user)
  - Is the ***only way*** a user program can perform privileged operations

System Call Overhead

- System calls are expensive and can hurt performance
- The system must do many things
  - Process is interrupted & computer saves its state
  - OS takes control of CPU & verifies validity of op.
  - **OS performs requested action**
  - OS restores saved context, switches to user mode
  - OS gives control of the CPU back to user process

Types of System Calls

- 5 categories:

1. Process Control
   - A running program needs to be able to stop execution
   - Normally or abnormally
   - If abnormally, dump of memory is created and taken for examination by a debugger
2. File Management
   - To perform operations on files
   - Create, delete, read, write, reposition, close
   - Many a times, OS provides an API to make these system calls

Types of System Calls

3. Device Management
   - Process usually requires several resources to execute
   - If available, access granted
   - Resources = devices
   - Eg: physical I/O devices attached
4. Information Management
   - To transfer information between user program and OS
   - Eg: time, date
5. Communication
   - Interprocess communication
     - Message passing model
     - Shared memory model

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

Example System Calls

```
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
```

- `int open(const char *pathname, int flags, mode_t mode);`
- `int close(int fd);`
- File descriptors
  - 0 stdin
  - 1 stdout
  - 2 stderr
- `ssize_t read(int fildes, void *buf, size_t nbyte)`
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to read
- `ssize_t write(int fildes, const void *buf, size_t nbyte);`
  - fildes: file descriptor
  - buf: buffer to write from
  - nbyte: number of bytes to write

Library Functions

- Functions that are a part of standard C library
- To avoid system call overhead use equivalent library functions
  - getchar, putchar vs. read, write (for standard I/O)
  - fopen, fclose vs. open, close (for file I/O), etc.
- How do these functions perform privileged operations?
  - They make system calls

- Many library functions invoke system calls indirectly
- So why use library calls?
- Usually equivalent library functions make fewer system calls
- non-frequent switches from user mode to kernel mode => less overhead

Unbuffered vs. Buffered I/O

- **Unbuffered**
  - Every byte is read/written by the kernel through a system call
- **Buffered**
  - collect as many bytes as possible (in a buffer) and read more than a single byte (into buffer) at a time and use one system call for a block of bytes

Buffered I/O decreases the number of read/write system calls and the corresponding overhead

File Descriptors

- File descriptor is an integer that uniquely identifies an open file of the process
- File descriptor table is the collection of integer array indices that are file descriptors in which elements are pointers to file table entries. One unique file descriptors table is provided in operating system for each process
- Read from stdin => read from fd 0: Whenever we write any character from keyboard, it is read from stdin through fd 0
- Write to stdout => write to fd 1: Whenever we see any output to the video screen it is written to stdout in screen through fd 1.
- Write to stderr => write to fd 2: Whenever we see any error to the video screen, it is written to stderr in screen through fd 2.

Open()

- Used to Open the file for reading, writing or both
- Syntax: int open (const char* Path, int flags);
  - Path: Path to file which is to be opened
    - Use Relative path if you are working in the same working directory as file
    - Otherwise, Absolute path, starting with ‘/’
  - Flags
    - O_RDONLY: read only,
    - O_WRONLY: write only,
    - O_RDWR: read and write,
    - O_CREAT: create file if it doesn’t exist,
    - O_EXCL: prevent creation if it already exists
- Returns a file descriptor

Create()

- Used to create a new empty file
- Syntax: int create(char *filename, mode_t mode)
  - Filename: name of the file which you want to create
  - Mode: Indicates permission of the new file
- returns first unused file descriptor (generally 3 because 0, 1, 2 fd are reserved) ; returns -1 when error

Difference between open() and create()

- create function *creates* files, but can not open existing file.
- Create is more of a legacy function now
- creat() is equivalent to open() with flags equal to O_CREAT|O_WRONLY|O_TRUNC

Close()

- Closes the file which pointed by fd
  - Frees the file descriptor
- Syntax: int close(int fd);
  - Fd: File Descriptor
- Returns 0 on success and -1 on error

Read()

- From the file indicated by the file descriptor fd, the read() function reads n bytes of input into the memory area indicated by buf.
- Syntax: size_t read (int fd, void* buf, size_t n);
  - fd: file descriptor
  - buf: buffer to read data from
  - n: length of buffer
- Returns:
  - Number of bytes read on success
    - File position advanced by this many bytes
  - 0 on reaching end of file
  - -1 on error

Write()

- Writes n bytes from buf to the file associated with fd
- Syntax: size_t write (int fd, void* buf, size_t cnt);
  - fd: file descriptor
  - buf: buffer to write data to
  - n: length of buffer
- Returns
  - Number of bytes written on success
  - 0
    - 0 written and fd is a regular file? Might return an error via errno (in a few slides)
      - Maybe no error will be detected
    - 0 written and fd isn’t regular? Results aren’t specified
  - -1 on error

Syscalls: Checking what went wrong

- Many syscalls will return an error value (often -1)
  - But how do you know what the error was?
- Syscalls will often set the value of *errno* on failure
  - Use errno like a variable
- Can check the value of errno to check the exact error and handle it
  - **EBADF** Bad file descriptor
  - **EDQUOT** Disk quota exceeded
  - **EACCES** Permission denied
  - etc.

Time and *strace*

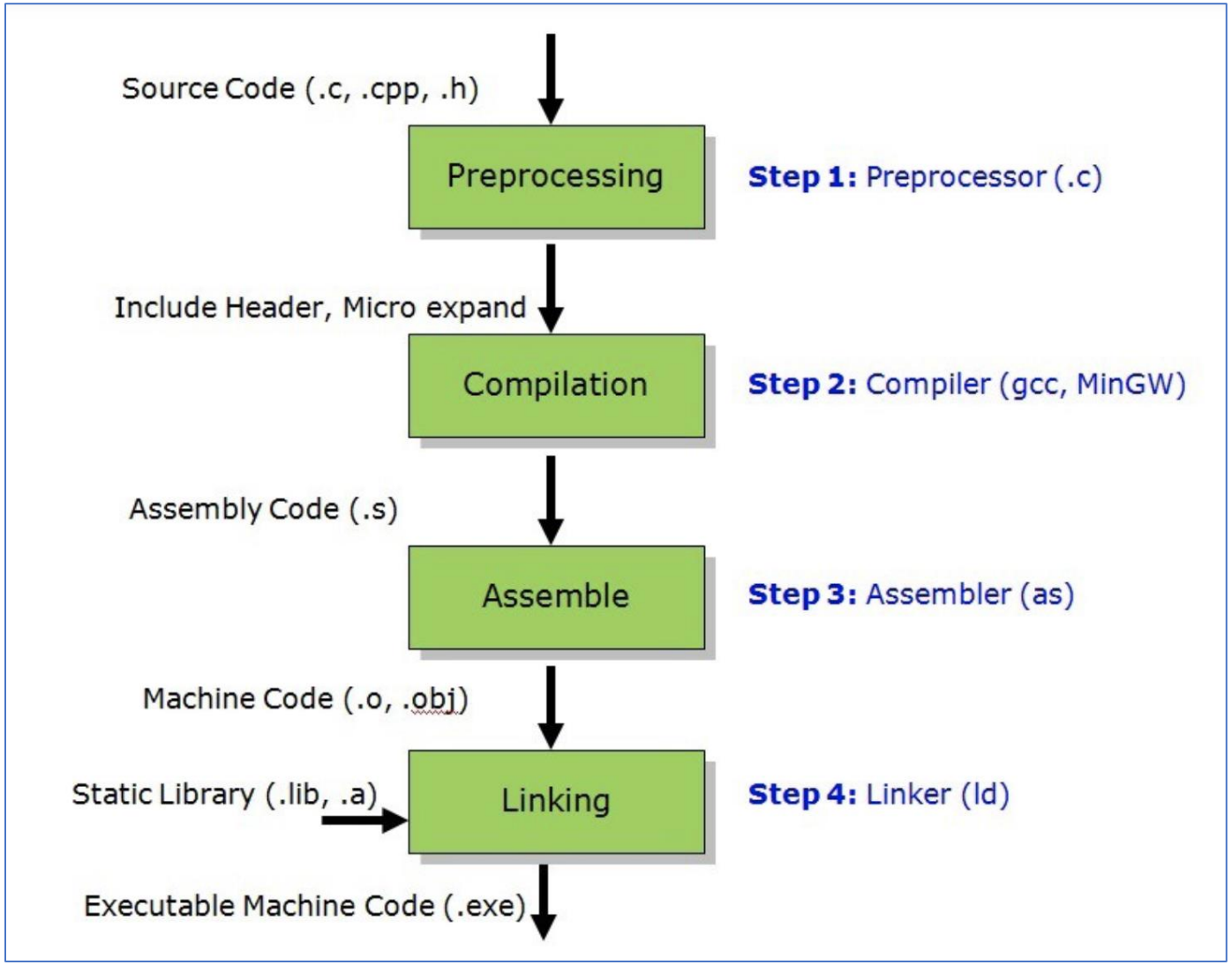
- **time [options] command [arguments...]**
  - Can give a very broad picture of performance
- Output:
  - –real 0m4.866s: elapsed time as read from a wall clock
  - –user 0m0.001s: the CPU time used by your process
  - –sys 0m0.021s: the CPU time used by the system on behalf of your process (in syscalls)
- **strace: intercepts and prints out system calls.**
  - –$ strace –c ./tr2b ‘AB’ ‘XY’ < input.txt
  - –$ strace –c ./tr2u ‘AB’ ‘XY’ < input.txt

fsanitize

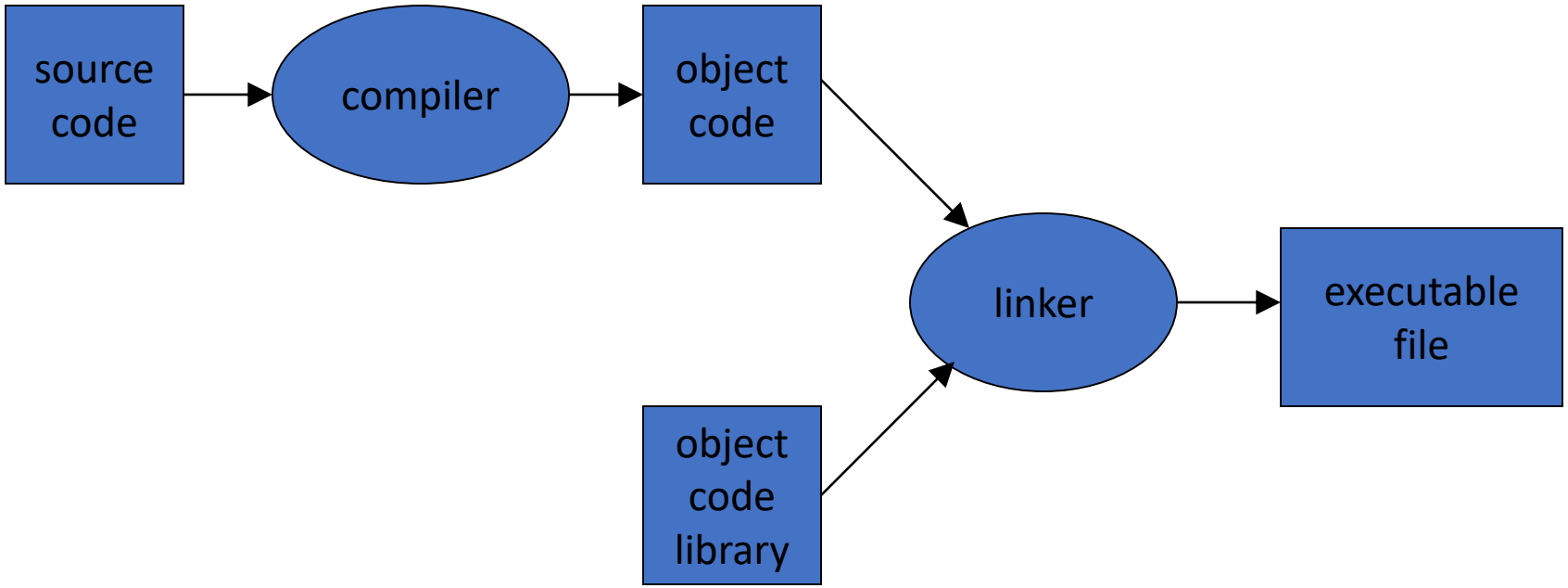
- Don’t ignore this slide, it’s very helpful
- Checks for memory issues (address mode) and runtime exceptions (undefined mode)
- Make sure to have /usr/local/cs/bin/ prepended to the path (works with the latest version of gcc)
- Compile using
  gcc –o out program-with-potential-memory-leak.c –fsanitize=address –static-libasan
  OR
  gcc –o out program-with-potential-undefined-behavior.c –fsanitize=undefined –static-libubsan
- Can use both at once if you want

CALL

- What happens between us writing our C program and it executing?
  - CALL
- Compiler
  - Takes in source, outputs assembly (still human readable)
- Assembler
  - Takes in assembly, outputs machine code
- Linker
  - Fills in the blanks, allowing program to use libraries
- Loader
  - Does different things on different platforms
  - Takes program from a file on disk to an executing process



http://binaryupdates.com/introduction-of-c/



A previously compiled collection of standard program functions

Static Linking

- Carried out only once
- If static libraries are used, the linker will copy all the modules referenced by the program into the executable
- Static libraries typically use a “.a” file extension

Dynamic Linking

- Allows a process to add, remove, replace or relocate object modules during its execution.
- If shared (dynamic) libraries are used:
  - Only copy a little reference information when the executable file is created
  - Complete the linking during loading time or running time
- Dynamic libraries typically use a “.so” file extension
  - .dll on Windows

Linking and Loading

- Linker collects procedures and links together the object modules into one executable program
- Why isn't everything written as just one big program, saving the necessity of linking?
  - Efficiency: if just one function is changed in a 100K line program, why recompile the whole program? Just recompile the one function and relink.
  - Use a new library without recompiling / redistributing
  - Avoid reading absolutely everything into memory
  - Allow programs to share the memory used by the library

Dynamic linking

- Unix systems: Code is typically compiled as a dynamic shared object (DSO)
- Dynamic vs. static linking resulting size

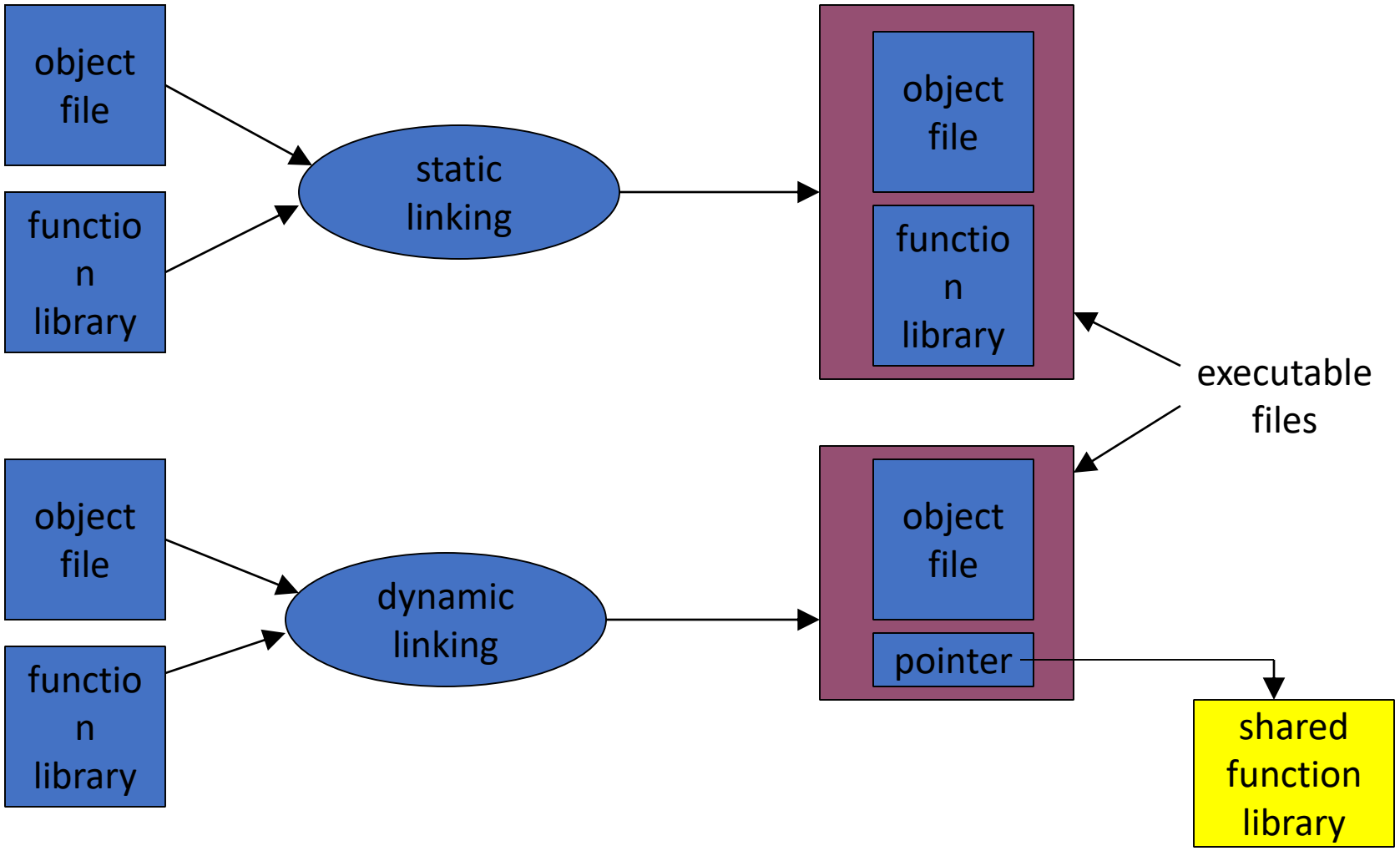
```
$ gcc -static hello.c -o hello-static
$ gcc hello.c -o hello-dynamic
$ ls -l hello
      80 hello.c
   13724 hello-dynamic
 1688756 hello-static
```

- Pros and cons?

Advantages of dynamic linking

- The executable is typically smaller
- When the library is changed, the code that references it does not usually need to be recompiled
- The executable accesses the .so at run time; therefore, multiple programs can access the same .so at the same time
  - Memory footprint amortized across all programs using the same .so

Smaller is more efficient



Disadvantages of dynamic linking

- (Slight) performance hit
  - Need to load shared objects (at least once)
  - Need to resolve addresses (once or every time)
  - Remember back to the system call assignment…
- What if the necessary dynamic library is missing?
- What if we have the library, but it is the wrong version?

ldd

- Usage: ldd ./my_program
- Prints out the shared libraries used by a program

```
[tylerd@lnxsrv07 ~]$ ldd ./sfrob
        linux-vdso.so.1 =>  (0x00007ffe749ec000)
        librt.so.1 => /lib64/librt.so.1 (0x00007fe70f395000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007fe70f191000)
        libpthread.so.0 => /lib64/libpthread.so.0
(0x00007fe70ef75000)
        libm.so.6 => /lib64/libm.so.6 (0x00007fe70ec73000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fe70ea5d000)
        libc.so.6 => /lib64/libc.so.6 (0x00007fe70e68f000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fe70f59d000)
```

Dynamic Linking

- Allows a process to add, remove, replace or relocate object modules during its 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
- Dynamic libraries are typically denoted by the .so (shared object) file extension in Unix system
  - .dll (dynamically linked library) on Windows
- Why we need shared libraries?

Dynamic Loading

- A mechanism to load shared library to memory at runtime
  - E.g. in a C program, use a line of code to load a new library when that line of code is being executed
- You still need to load a shared library
- What’re the advantages?
  - Allow start up in the absence of some library
  - Avoid linking unnecessary libraries

Big Picture

- 1. Create a static/shared library
  - Use ar (for static) or gcc (for dynamic)
- 2. Link that library when you compile your program
  - Use gcc (note all the flags)
- 3. If necessary, use dynamic load in your program
  - dlopen(), dlsym(), dlclose()
- 4. Automate the process
  - make

Dynamic Loading

- A sample program
- On success, **dlopen**() returns a non-NULL handle for the loaded library. On error, returns NULL.
- **dlsym**() returns NULL for error
- On success, **dlclose**() returns 0; on error, it returns a nonzero value
- Saves error to **dlerror(**)

```
void *handle;
double (*cosine)(double);
char *error;
handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
if (!handle) {
    // Handle error
    // exit(1);
}
cosine = dlsym(handle, "cos");
if ((error = dlerror()) != NULL)  {
    // Handle error
    // exit(1);
}
printf ("%f\n", (*cosine)(2.0));
dlclose(handle);
```

Makefile Example

```
# Makefile - A Basic Example
all : shop  #usually first
shop : item.o shoppingList.o shop.o
        g++ -g -Wall -o shop item.o shoppingList.o shop.o
item.o : item.cpp item.h
        g++ -g -Wall -c item.cpp
shoppingList.o : shoppingList.cpp shoppingList.h
        g++ -g -Wall -c shoppingList.cpp
shop.o : shop.cpp item.h shoppingList.h
        g++ -g -Wall -c shop.cpp
clean :
        rm -f item.o shoppingList.o shop.o shop
```

Tab here →

A Rule

Comments

Targets

Prerequisites

Dependency Line

Commands

Makefile Structure and Logic

- A *target* is usually the name of a file that is generated by a program
- A *prerequisite* is a file that is used as input to create the target
- A *recipe* is an action that make carries out
- Before make can fully process this rule, it must process the rules for the files that it depends on (the prerequisites)
- The recipe will be carried out if
  - Any file named as prerequisites is more recent than the target file
  - If the target file does not exist at all

Executing Makefile

- When you run `make` command, *the first target* in your Makefile
  - By default, it will look for a file called “Makefile”, use –f to override
- To run a specific target instead of the first, use $ make [target name]
  - Example: make install
- .PHONY target
  - By default, make thinks that the target name is a file
  - Imagine you have a rule `make clean` to delete files, yet you have a file named “clean” in your directory
  - Now `make clean` will tell you nothing to make (why?)
  - Use phony target to enforce `make clean`

Makefile Variable

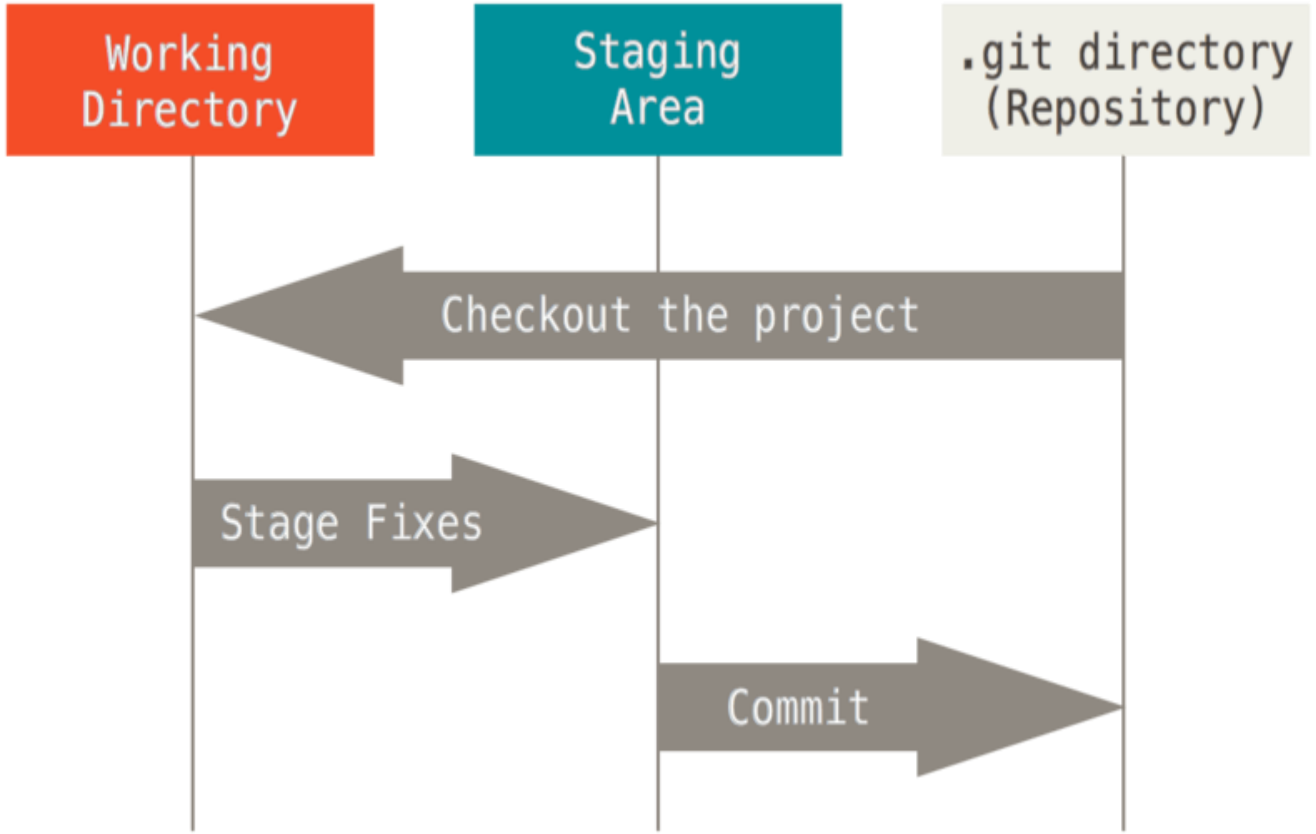
- [name] = [value]
- To use it later, use $(name)
  - It is conventional to have variables like CC, CFLAGS, etc.
- Automatic variables
  - $@: Target name
  - $<: First prerequisite
  - $?: All prerequisites newer than the target
  - $^: All prerequisites

Git Basics

- Check out some code
  - Get a copy of the files
    - Could be from your machine
    - Could be from someone else’s
- Make your changes
- Stage the changes you want to commit
  - Which changes you want to track
- Commit your staged changes
  - Like hitting “save”
- Share your changes

Git States

- Files can exist in three main states
  - Modified
    - File changed but not committed
  - Staged
    - Modified and marked to be committed
  - Committed
    - Safely stored in database



git-scm.org

Git Commands

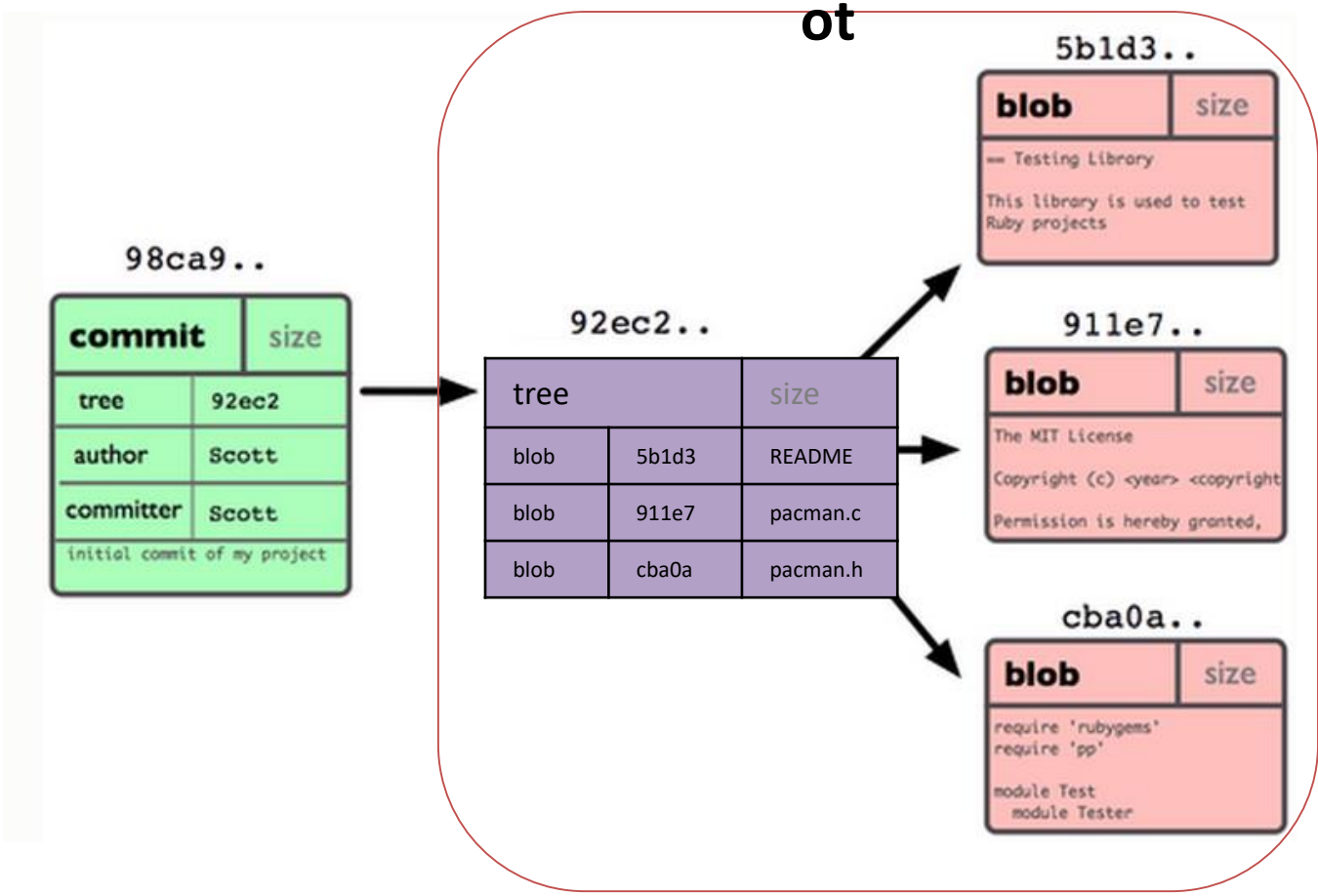
- Repository creation
  - $ git init (Create a new repository)
  - $ git clone (Create a copy of an existing repo)
- Branching
  - $ git branch <new_branch_name>
  - $ git checkout <tag/commit> -b <new_branch_name>
- Commits
  - $ git add (Stage modified/new/deleted files)
  - $ git commit (Save changes to repository)

Git Commands

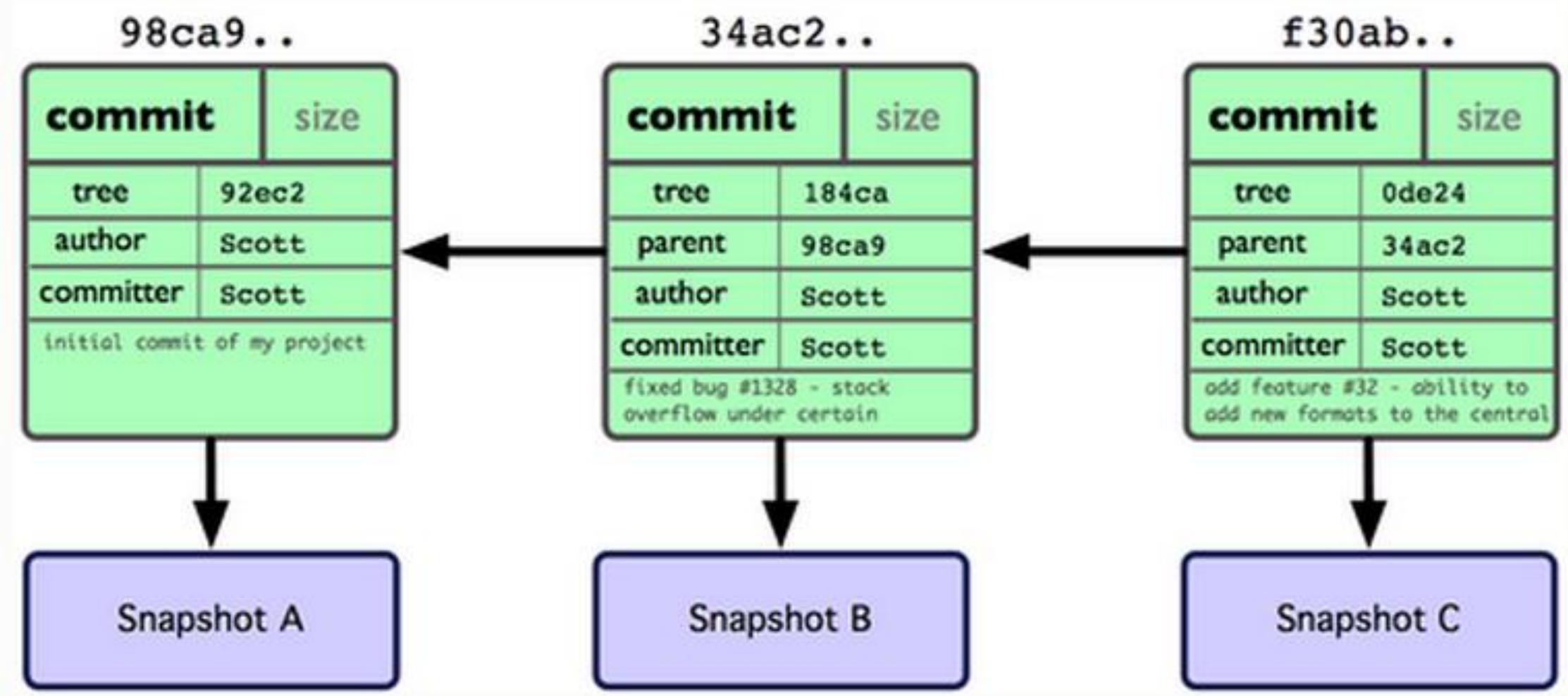
- Getting information
  - $ git status (Shows state of modified files, new files, etc.)
  - $ git diff (Compare different versions of files)
  - $ git log (Shows history of commits)
  - $ git show (Shows object in the repository)
- Help
  - $ git help

Git Repo Structure

- A commit corresponds to a snapshot
- Snapshot is a picture of your repo at the time you commit
  - If file is unchanged since last snapshot, just point at its last version
- Tree
  - Think a “collection of files”
- Blob
  - A version of a file
- Checksum
  - Run SHA1 on object to get an identifier
  - This is how git refers to the object

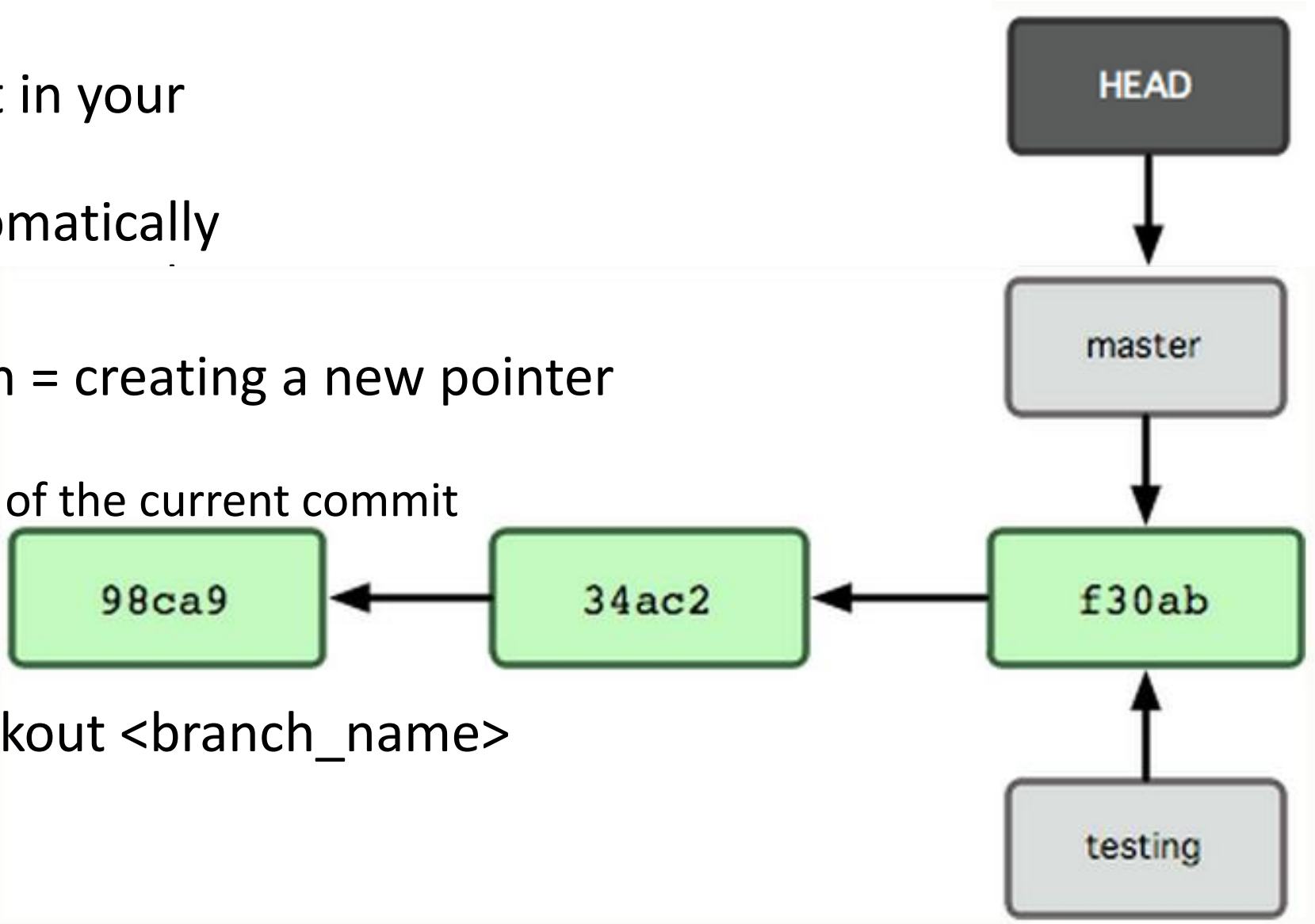


After Two More Commits



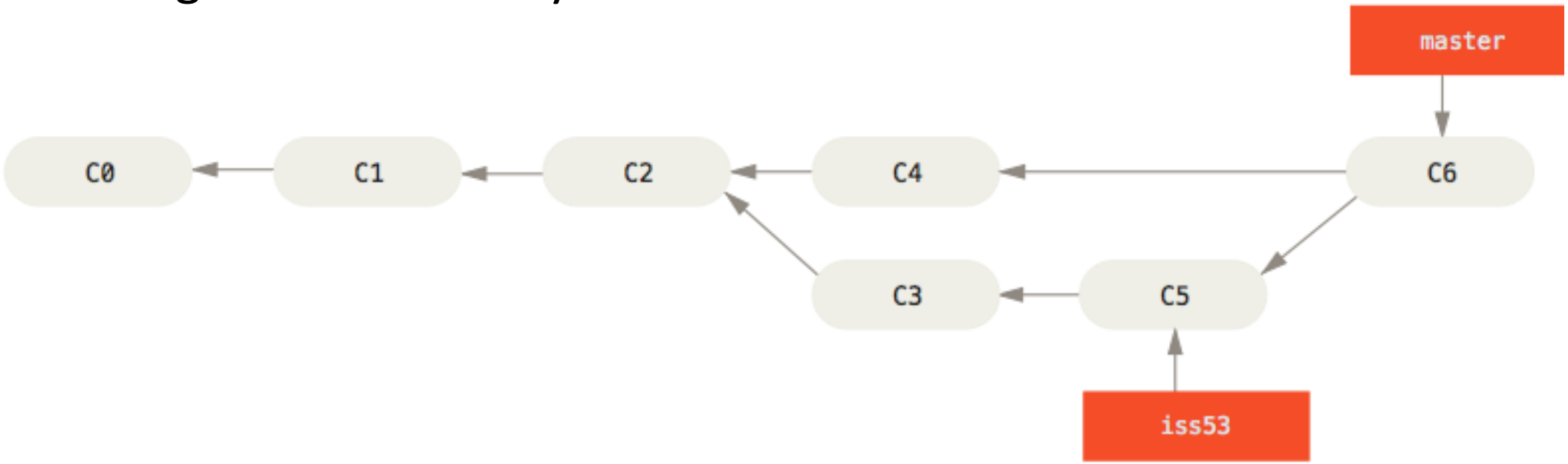
What is a branch?

- A pointer to a commit in your repo + its history
- “Master” branch automatically created when repo is
- Creating a new branch = creating a new pointer
- $ git branch testing
  - Creates a branch off of the current commit
    - Aka HEAD
- Switch with $ git checkout <branch_name>



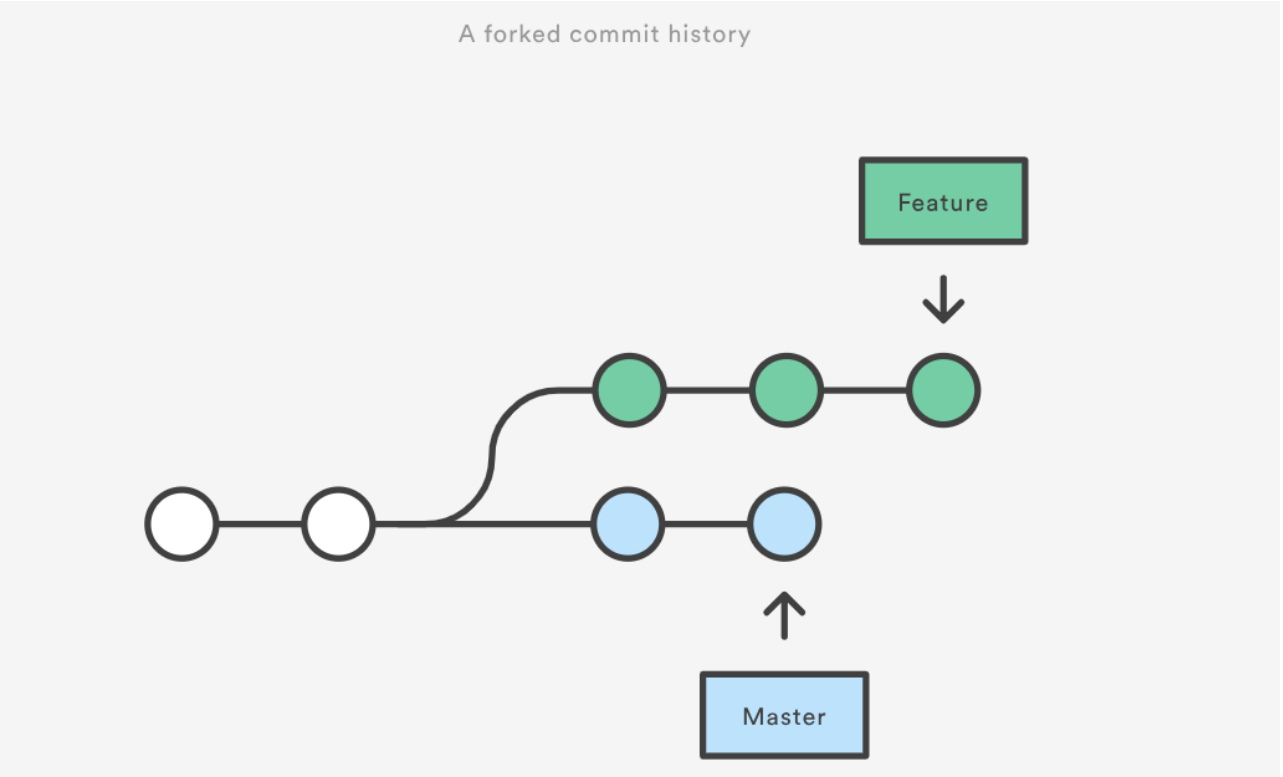
Git Merge

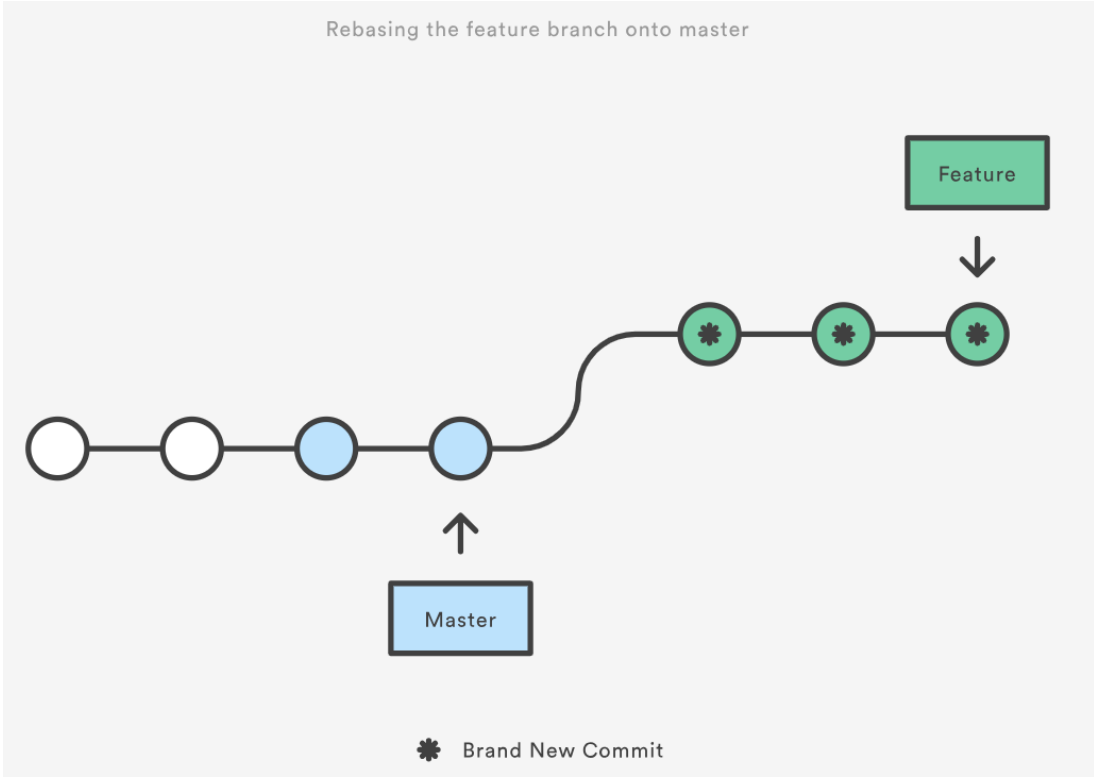
- Puts a merge commit in your history
- Created via 3 way merge between common ancestor and snapshots C4 and C5
- Con: Merge commits everywhere



Git Rebase

- Moves the “base” the branch you’re integrating with from the common ancestor to HEAD
- Cleaner, but you rewrite history





Remote Repositories

- You might want to interact with a repository elsewhere
  - On the local network
  - On GitHub
- Good for collaboration
- A bit of an offsite backup
- Git clone automatically adds an “origin” repository
- $ git remote
  - List info about remotes
  - Lets you manage them
    - Add
    - Remove
  - $ git remote show origin
    - Shows info about origin

Remote Branches

- Branches that correspond to remote branches
- Git helps you keep them in sync
- Take the form <remote>/<branch>
  - E.g. origin/master
- Can make new branches that track remote as well
  - $git checkout -b <branch> <remote/branch>
  - $git checkout --track <remote>/<branch?

Working With Remotes

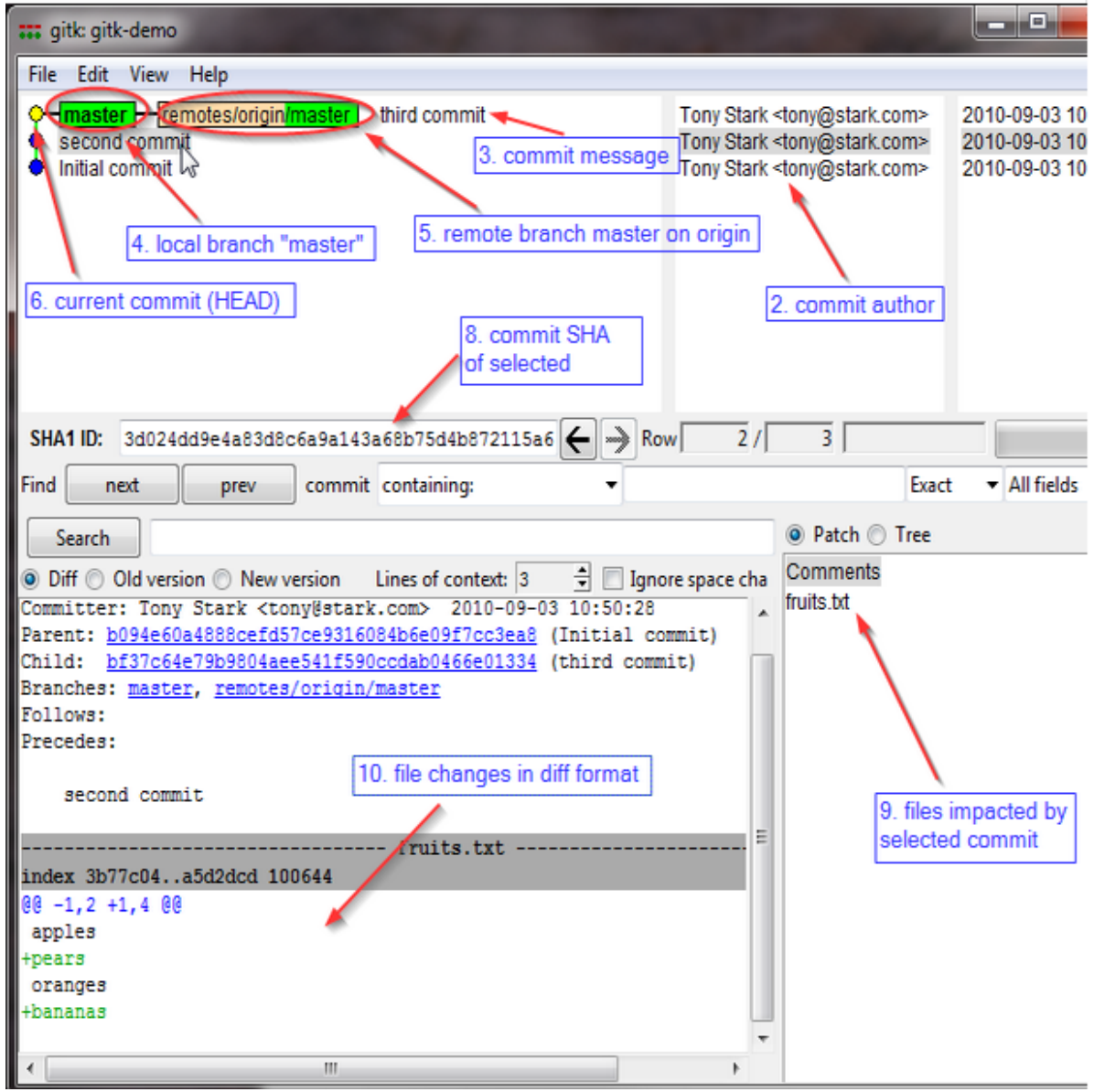
- $git fetch <remote>
  - Pull all the info about <remote> to your local
    - Doesn’t do any merging
  - <remote> is origin if not specified
- $git pull <remote> <branch>
  - $git pull origin master
  - Grabs changes from the remote, and merges them into **current** branch
    - Like git fetch followed by git merge
- You’ve made changes locally, how do you send them to the remote?
  - $ git push
- Sends all your changes to the remote repository
  - If there is a conflict, will alert you
  - Need to resolve conflicts locally, then reattempt push
- Push might be restricted
  - Only certain users can push to a repo/branch

More Git Commands

- Reverting
  - $ git checkout HEAD main.cpp
    - Gets the HEAD revision for the working copy
  - $ git checkout -- main.cpp
    - Reverts the changes in the working copy
  - $ git revert
    - Reverts a commit with a new commit
- Cleaning up untracked files
  - $ git clean
- Tags
  - Human readable pointers to specific commits
  - $git tag -a v1.0 -m ‘Version 1.0’
    - Names the current HEAD commit as v1.0

GitK

- Git is great, but the command line can be a real pain
- GitK helps
  - Visualize commit graphs
  - Understand repo structure
- Here's a tutorial
- And some missing documentation
- Other GUIs exist!
  - Github Desktop, Git Kraken, and Git Tower all popular
- Github does some of this too



Communication Over the Internet

- What type of guarantees do we want?
  - **Confidentiality**
    - Message secrecy
  - **(Data) Integrity**
    - Message consistency
  - **Authentication**
    - Identity confirmation
  - **Also authorization**
    - Specifying access rights to resources

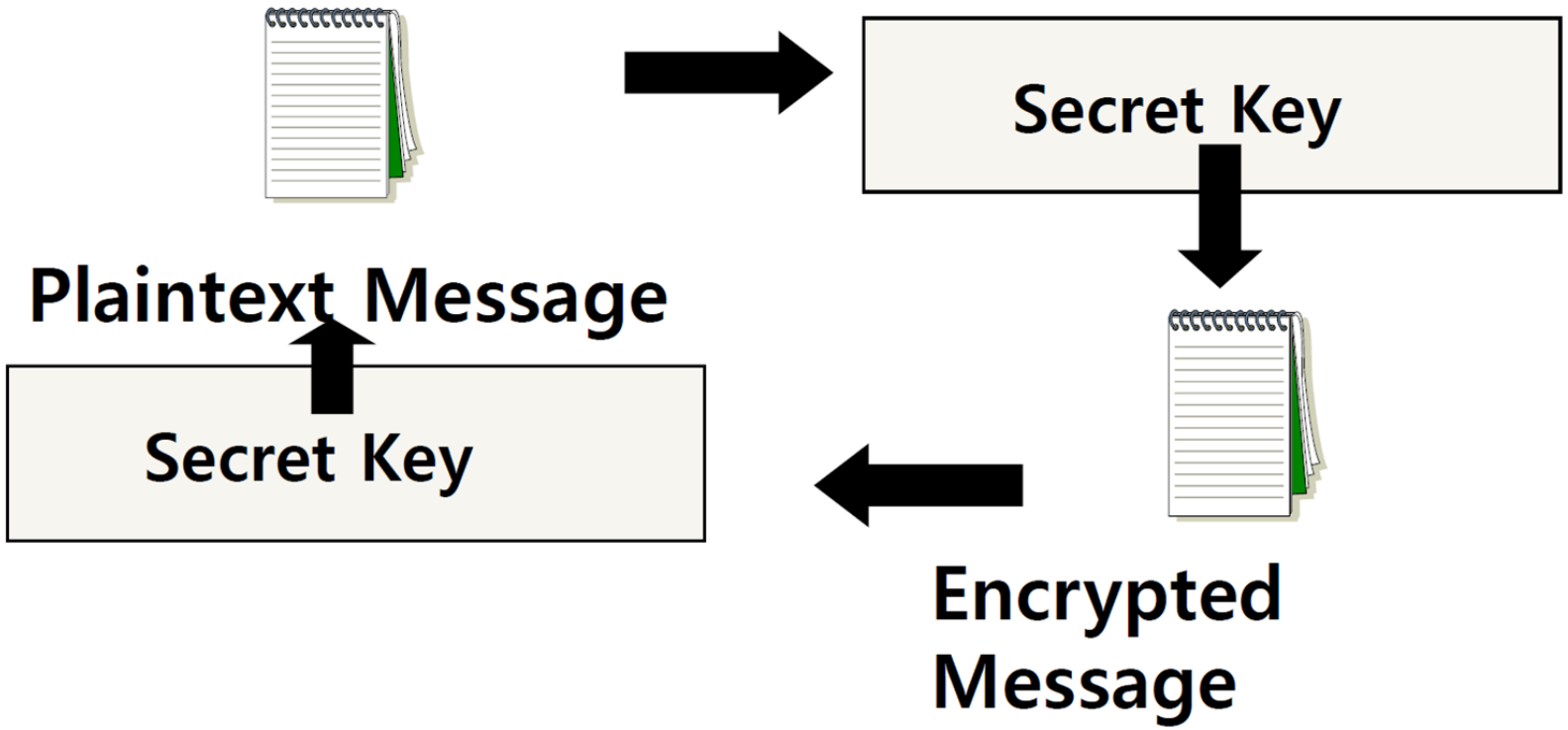
Encryption Types

- **Symmetric Key Encryption**
  - a.k.a 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
    - Only creator knows the relation. Private key cannot be derived from public key
  - Data encrypted with public key can only be decrypted by private key and vice versa
  - Public key can be seen by anyone
  - Never publish private key!!!

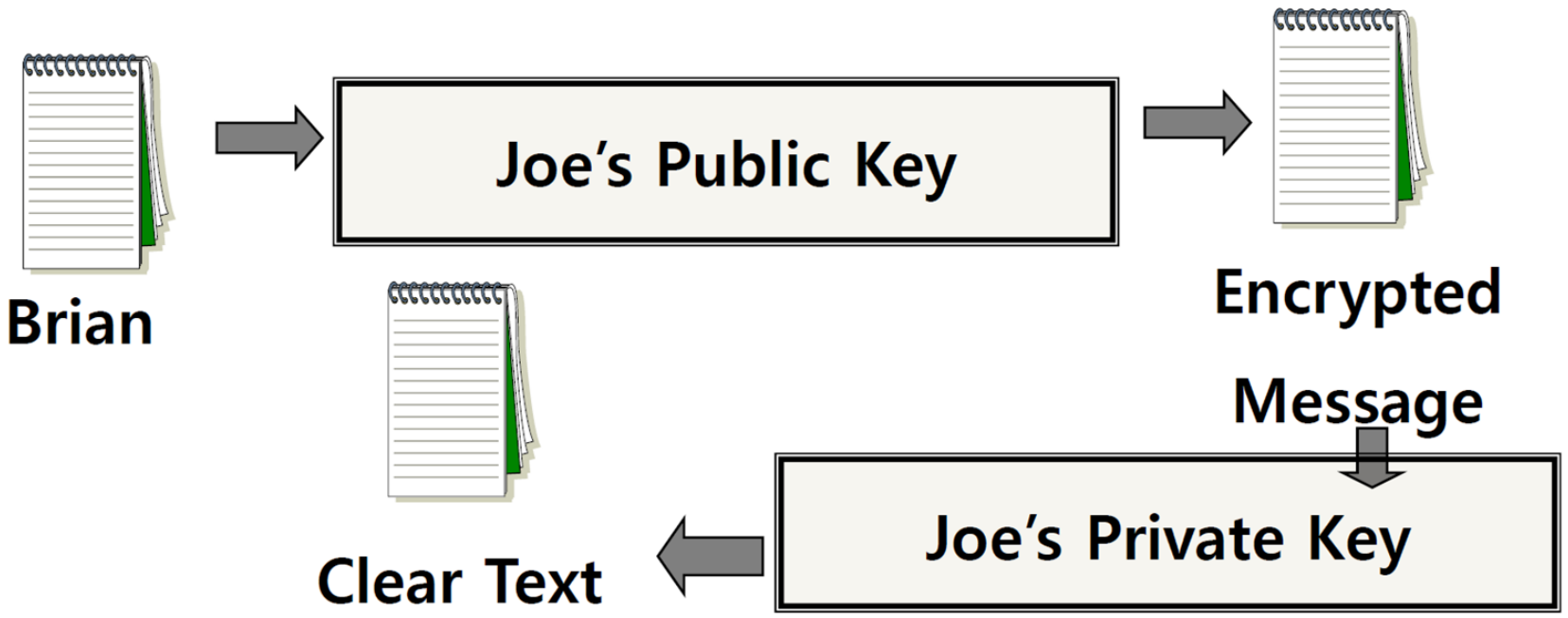
Secret Key (symmetric) Cryptography

- A single key is used to both encrypt and decrypt a message



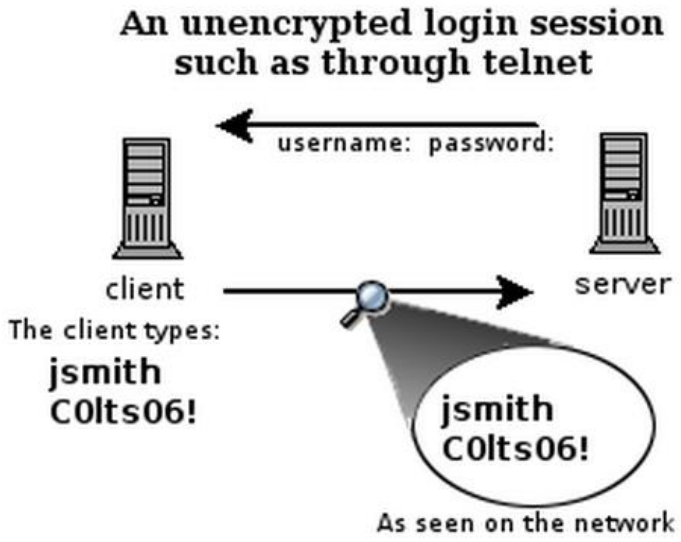
Public Key (asymmetric) Cryptography

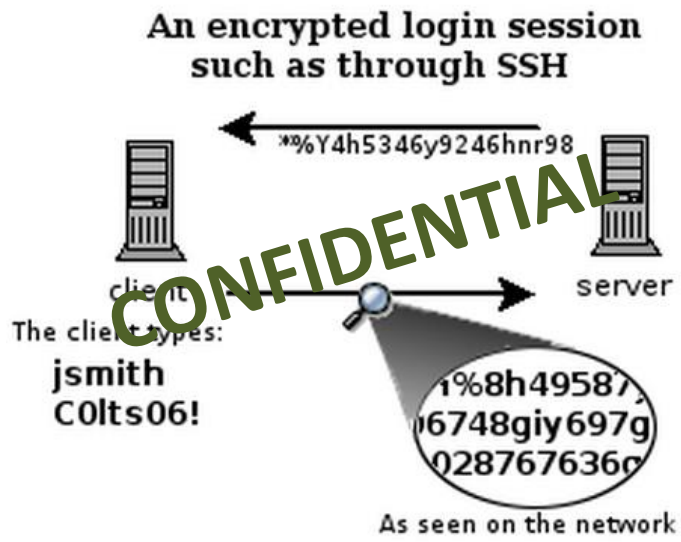
- Two keys are used: a public and a private key. If a message is encrypted with one key, it has to be decrypted with the other.



What is SSH?

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





High-Level SSH Protocol

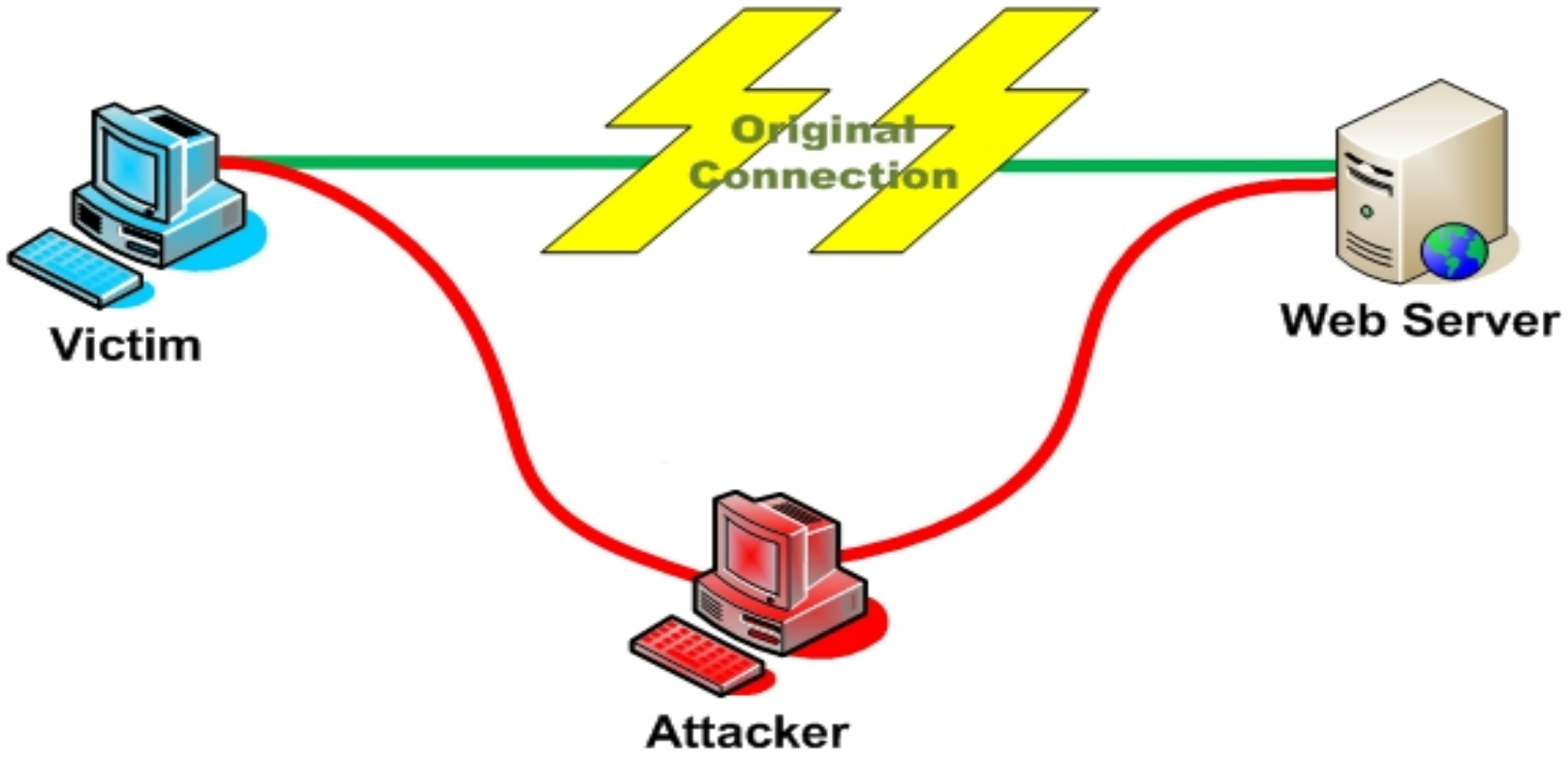
- Client ssh's to remote server
  - `$ ssh username@somehost`
  - If first time talking to server -> host validation

The authenticity of host 'somehost (192.168.1.1)' can't be established.
RSA key fingerprint is 90:9c:46:ab:03:1d:30:2c:5c:87:c5:c7:d9:13:5d:75.
Are you sure you want to continue connecting (yes/no)? **yes**
Warning: Permanently added 'somehost' (RSA) to the list of known hosts.

  - ssh doesn't know about this host yet
  - shows hostname, IP address and fingerprint of the server's public key, so you can be sure you're talking to the correct computer
  - After accepting, public key is saved in ~/.ssh/known_hosts

Host Validation

- Next time client connects to server
  - Check host's public key against saved public key
  - If they don't match



Host Validation (cont'd)

- Client asks server to prove that it is the owner of the public key using **asymmetric encryption**
  - Encrypt a message with public key
  - If server is true owner, it can decrypt the message with private key
- If everything works, host is successfully validated

AUTHENTICATION

Session Encryption

- Client and server agree on a **symmetric encryption** key (session key)
- All messages sent between client and server
  - encrypted at the sender with session key
  - decrypted at the receiver with session key
- anybody who doesn't know the session key (hopefully, no one but client and server) doesn't know any of the contents of those messages

User Authentication

- **Password-based authentication**
  - Prompt for password on remote server
  - If username specified exists and remote password for it is correct then the system lets you in
- **Key-based authentication**
  - Generate a key pair on the client
  - Copy the public key to the server (~/.ssh/authorized_keys)
  - Server authenticates client if it can demonstrate that it has the private key
  - The private key can be protected with a passphrase
  - Every time you ssh to a host, you will be asked for the passphrase (inconvenient!)

ssh-agent (passphrase-less ssh)

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

Server Steps

- **Generate public and private keys**
  - $ `ssh-keygen` (by default saved to ~/.ssh/is_rsa and id_rsa.pub) – don't change the default location
- **Create an account for the client on the server**
  - $ `sudo useradd -d /home/<homedir_name> -m <username>`
  - $ `sudo passwd <username>`
- **Create .ssh directory for new user**
  - $ `cd /home/<homedir_name>`
  - $ `sudo mkdir .ssh`
- **Change ownership and permission on .ssh directory**
  - $ `sudo chown -R username .ssh`
  - $ `sudo chmod 700 .ssh`

Client Steps – Make logins convenient

- **Generate public and private keys**
  - `$ ssh-keygen`
- **Copy your public key to the server for key-based authentication (~/.ssh/authorized_keys)**
  - `$ ssh-copy-id -i UserName@server_ip_addr`
- **Add private key to authentication agent (ssh-agent)**
  - `$ ssh-add`
- **SSH to server**
  - `$ ssh UserName@server_ip_addr`
  - `$ ssh -X UserName@server_ip_addr` (X11 session forwarding)
- **Run a command on the remote host**
  - `$ xterm`, `$ gedit`, `$ firefox`, etc.

How to Check IP Addresses

- $ `ifconfig`
  - configure or display the current network interface configuration information (IP address, etc.)
- $ `hostname -I`
  - gives the IP address of your machine directly
- $ `ping <ip_addr>`(**p**acket **in**ternet **g**roper)
  - Test the reachability of a host on an IP network
  - measure round-trip time for messages sent from a source to a destination computer
  - Example: $ ping 192.168.0.1, $ ping google.com

Steps for Generating a Digital Signature

Ensures data integrity (document was not changed during transmission)

SENDER:

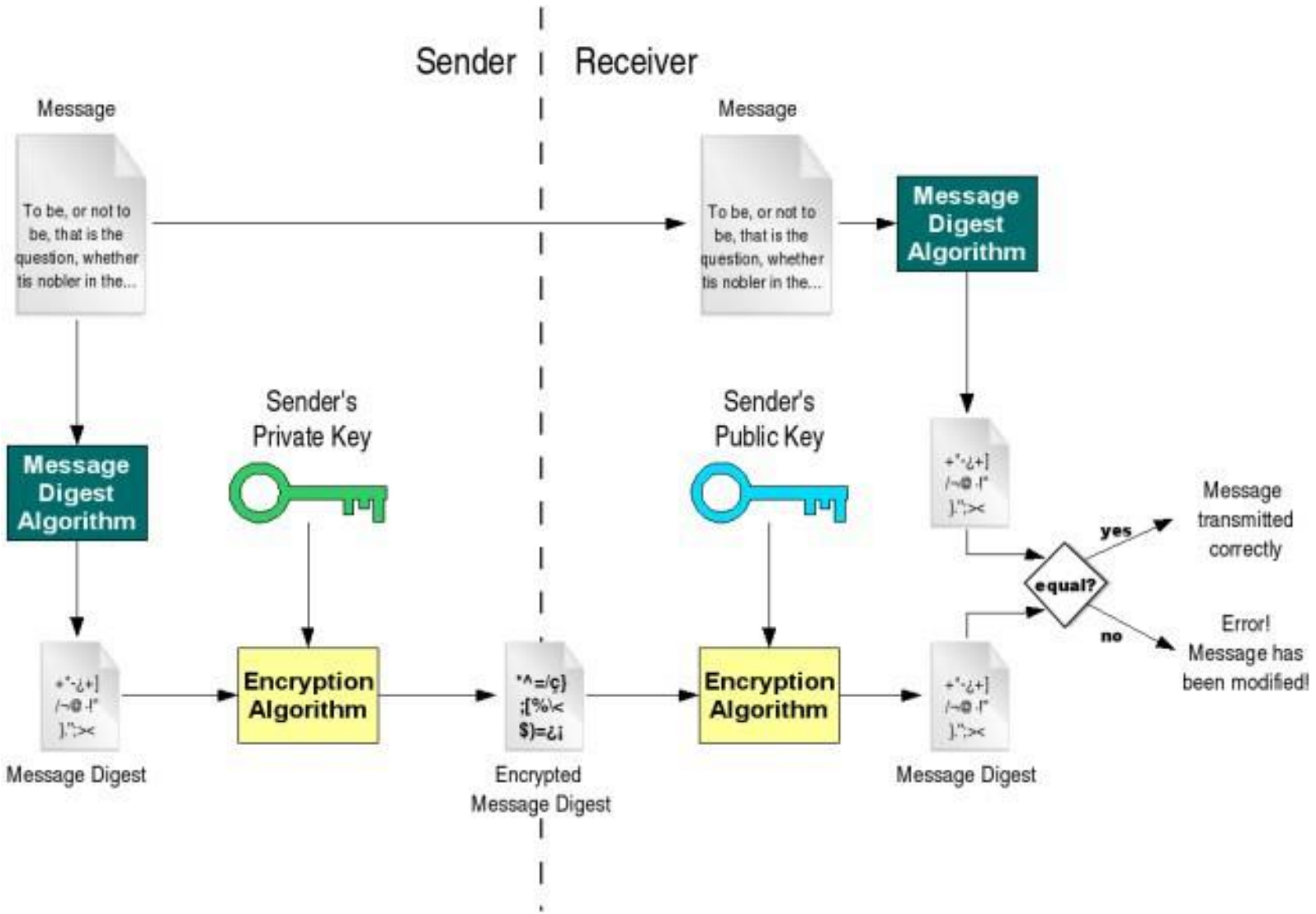
1) Generate a *Message Digest*
   - The message digest is generated using a set of hashing algorithms
   - A message digest is a 'summary' of the message we are going to transmit
   - Even the slightest change in the message produces a different digest
2) Create a Digital Signature
   - The message digest is encrypted using the sender's *private* key. The resulting encrypted message digest is the *digital signature*
3) Attach digital signature to message and send to receiver

Steps for Generating a Digital Signature

RECEIVER:

1) Recover the *Message Digest*
   - Decrypt the digital signature using the sender's public key to obtain the message digest generated by the sender
2) Generate the Message Digest
   - Use the same message digest algorithm used by the sender to generate a message digest of the received message
3) Compare digests (the one sent by the sender as a digital signature, and the one generated by the receiver)
   - If they are not *exactly the same* => the message has been tampered with by a third party
   - We can be sure that the digital signature was sent by the sender (and not by a malicious user) because *only* the sender's public key can decrypt the digital signature and that public key is proven to be the sender's through the certificate.
   - If decrypting using the public key renders a faulty message digest, this means that either the message or the message digest are not exactly what the sender sent.

Digital Signature



Detached Signature

- Digital signatures can either be *attached* to the message or *detached*
- A detached signature is stored and transmitted separately from the message it signs
- Commonly used to validate software distributed in compressed tar files
- You can't sign such a file internally without altering its contents, so the signature is created in a separate file

Revision Selection - Ancestry References

Suppose we have the following commits:

```
$ git log --pretty=format:'%h %s' --graph
*   2e25043 Merge pull request #18 from ...
|\
| * 4950521 fix sim by normalizing SNP
columns
| * 69402cd generate g effects
|/
* c455717 tabulate_output.py
* f3fe695 Merge pull request #17 from ...
```

A caret **^** at the end of a reference refers to the parent of that commit. e.g. **c455717^** will refer to **f3fe695**

For merge commits such as **2e25043**, **2e25043^** will refer to its first parent, **c455717** while **2e25043^2** will refer to its second parent **4950521**.

Moreover, since HEAD points to **2e25043**, **HEAD^** and **2e25043^** are equivalent.

A tilde ~ also refers to the first parent, so HEAD^ and HEAD~ are equivalent.

HEAD~2 refers to the first parent of the first parent of HEAD, and the same pattern goes for HEAD~3 etc.

Revision Selection - Commit Ranges

```
git log master..experiment
```

will show commits in experiment not reachable from master
so the output will be

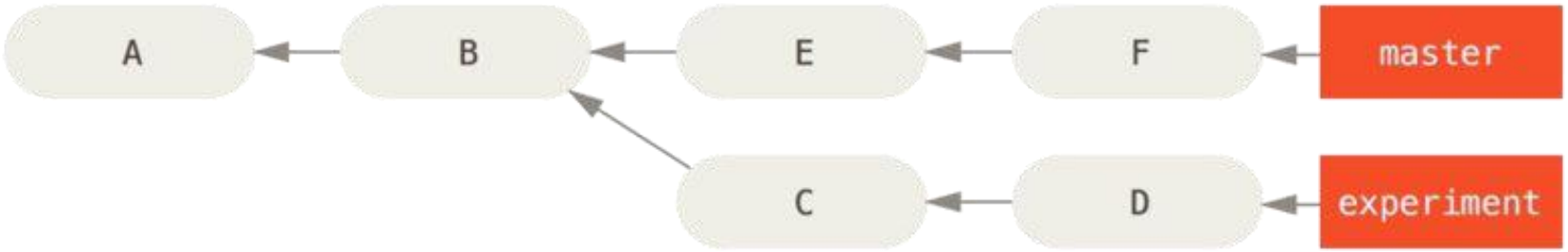
```
D
C
```

On the other hand,

```
git log experiment..master
```

will output

```
F
E
```



Rewriting History

In order to rewrite the last few commits, use the git interactive rebase, e.g.

```
git rebase -i HEAD~3
```

will rebase the last 3 commits onto the 4th oldest commit.

You will see an editor open up with texts like the ones in the next

slide. Beware the the commits are listed in the reverse order to that of

git log.

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
```

Three Trees

HEAD

Last commit snapshot, next parent. HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch.

Index

Proposed next commit snapshot. The index is your proposed next commit. We've also been referring to this concept as Git's "Staging Area" as this is what Git looks at when you run git commit.

Working Directory

The actual directories with files that you can modify with an editor.

Git Plumbing Commands

Plumbing commands refer to a set of git commands that do low-level work and are not usually used for ordinary purposes.

For example, **git hash-object** is a plumbing command:

```
echo 'test content' | git hash-object -w --stdin
```

--stdin indicates taking input from stdin, and -w indicates writing the content into a new file in .git/objects

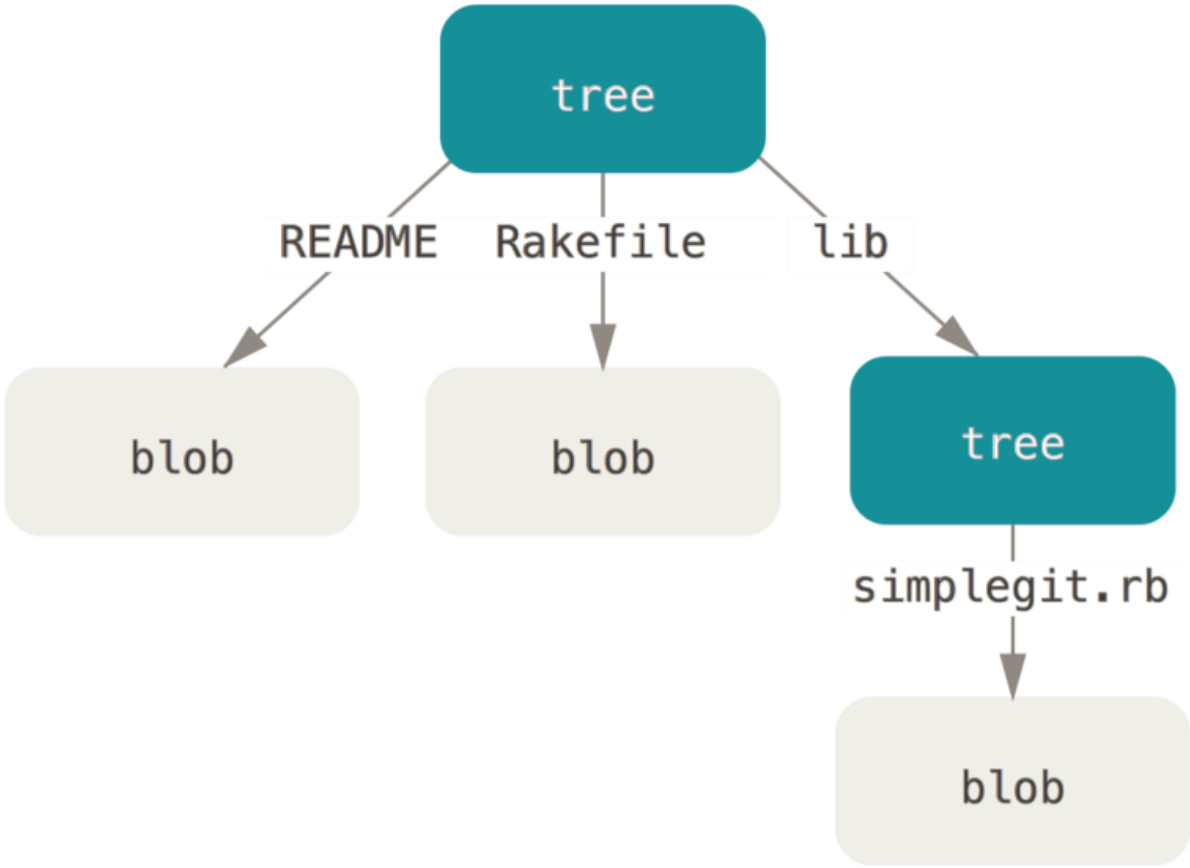
Git Objects

```
echo 'version 1' > test.txt
git hash-object -w test.txt
# returns a hash, say 83baae61804e65cc73a7201a7252750c76066a30

echo 'version 2' > test.txt
git hash-object -w test.txt
# returns another hash, say 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

find .git/objects -type f
# .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
# .git/objects/83/baae61804e65cc73a7201a7252750c76066a30
# ...

git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30
# prints version 1
git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
# prints version 2

git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
# prints "blob", the type of the file, due to the -t option
# blob is the leaf of the directory tree, representing a regular file.
```

Tree Objects

Another type of git objects is a tree object, which represents directory entries.

For example, each commit will point to a tree object, which is the top directory of the repository when the commit snapshot is taken.



Commit Objects

Each commit is also stored as a git object, which will contain information such as the tree object is points to, the parent commit(s), author, commit message, etc.

```
$ git cat-file -p 02250b1e77c88a20fba80bad2a76bb737d2d59e9
                  tree 671ea1871bdeb5f27881428c00abb4cc12a3238c
                  parent 55bb5940a549a475c768b503cb1b76a6f029d444
                  author Aaron <rustinante@gmail.com> 1574341950 -0800
                  committer Aaron <rustinante@gmail.com> 1574341950 -0800
                  b1 4
```

Git References

Inside .git/refs/ there are files whose names are aliases to commit hashes.

For example, .git/refs/heads/master will contain the commit hash of the commit pointed to by the master branch.

Another set of references are the remotes.

For example, refs/remotes/origin/master will contain the commit hash pointed to by the master branch in the remote named origin.

Git Refspec

When we do something like **git remote add origin some-url**, an entry is created in the **.git/config** file:

```
[remote "origin"]
        url = some-url
        fetch = +refs/heads/*:refs/remotes/origin/*
```

The fetch has the format +<source>:<destination>, where <source> is the references on the remote, and <destination> is where those references will be tracked locally.
The + sign means update the local references even if the remote branch update is not a fast forward, e.g. if originally we have commits **A <-- B (origin/master)(master)**, then somebody viciously changed history on the remote to A <-- C, the + sign says this is okay and update the local references to be

Topological Order

A **topological order** on a directed acyclic graph (DAG) G = (V, E) is a **total order** on all of its vertices such that if there is a directed edge from **v1** to **v2**, then **v1** < **v2**.

We can view a topological ordering/sorting on G as arranging the vertices on a horizontal line such that all edges go from left to right.

Depth-First Search (DFS) for Both Directed and Undirected Graphs

```
dfs(G = (V, E), s):
    for each vertex u in V:
        u.color = white
        u.discoverer = None
    time = 0
    for each vertex u in G.adj[s]:
        if u.color == white:
            dfs_visit(u)
```

```
dfs_visit(u):
    time = time + 1
    u.discover_time = time
    u.color = gray
    for each v in G.adj[u]:
        if v.color == white:
            v.discoverer = u
            dfs_visit(v)
    u.color = black
    time = time + 1
    u.finish_time = time
```

This is the pseudo code.

- white means undiscovered
- gray means being processed
- black means finished processing

G.adj[**u**] is the set of vertices reachable from **u**.

A stack can (and probably should) be used to implement DFS instead of recursion.

In fact, a stack should be used in Python to implement DFS due to the interpreter's limit on recursion depth.

Python DFS Implementation Using a Stack

Depending on your need, each element on the stack can be a vertex coupled with some extra info.

For example, if you want to keep track of the path, something like this will help:

```
stack = [(vertex, [vertex])]
visited = set()
while stack:
    v, path = stack.pop()
    visited.add(v)
    do something...
    for child in v.adjacent_vertices:
        if child not in visited:
            stack.append((child, path + [child]))
        do something...
```