Final Review

CS 35L
Spring 2018 - Lab 3

Final Information

- Thursday, June 14th, 3pm-6pm
- Boelter 5280 (NOT the usual room)
- Open book and open note
  - No laptops, calculators, smartphones, smartwatches, smart-glasses, etc.
- 50% of course grade (from syllabus)

Final Information

- Questions will cover assignments 1-9
  - Conceptual multiple choice/short answer
  - Code writing exercises in several languages
  - No questions on specific emacs/vim commands, everything else is fair game
- All questions that require referencing documentation are written to be answerable using only the material on this quarter's slides.

About these review slides

- Not comprehensive
  - Meant to give you a review of some of the concepts we covered in the course
- Conceptual understanding of material is more important than memorization
  - Also need to be comfortable writing code
- Questions to get you thinking about course material are posed throughout the review slides
  - Strive to be able to confidently answer all questions

Assignment 1

Unix

GNU/Linux

- Open-source operating system
  - **Kernel**: core of operating system
    - Allocates time and memory to programs
    - Handles file system and communication between software and hardware
  - **Shell**: interface between user and kernel
    - Interprets commands user types in
    - Takes necessary action to cause commands to be carried out
  - **Programs**

Files and Processes

- Everything is either a <u>**process**</u> or a <u>**file**</u>:
  - **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: Shell

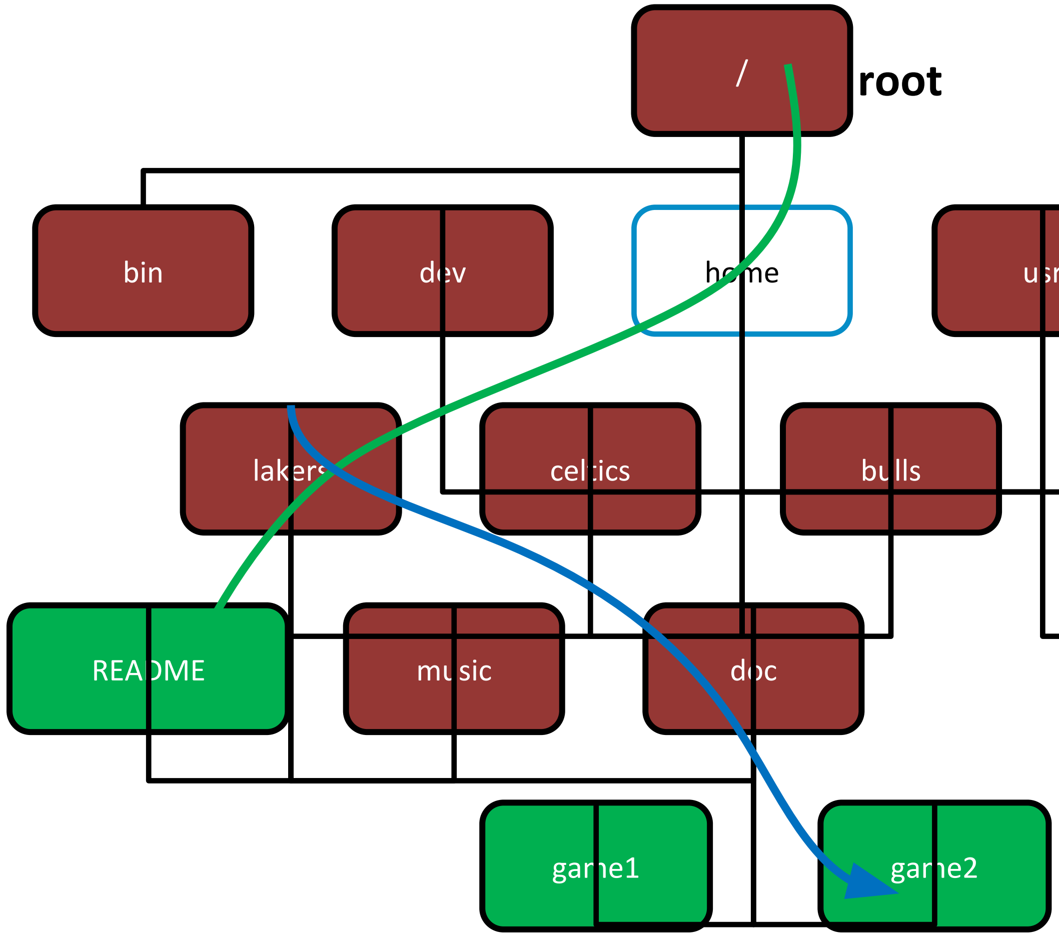
Some of the CLI utilities from you should be familiar with:

- `pwd`
- `cd`
- `mv`
- `cp`
- `rm`
- `mkdir`
- `rmdir`
- `ls`
- `ln`
- `touch`
- `find`
- `which`
- `man`
- `ps`
- `kill`
- `diff`
- `wget`
- `tr`
- `wc`
- `grep`
- `and others...`

The Basics: Shell

- How do I find where files are on the system?
- How do I find out what options are available for a particular utility?
- When is a file a file and when is it a process?
- What types of links are there?

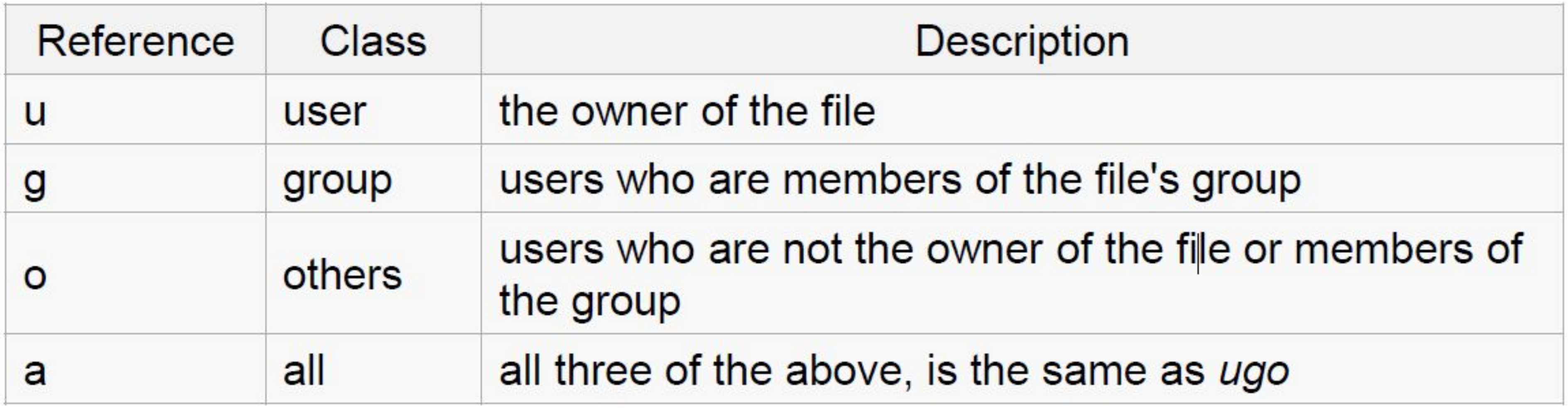
Absolute Path vs. Relative Path



Current directory: home   What are the differences between absolute and relative paths?

Linux File Permissions

- chmod
  - read (r), write (w), executable (x)
  - User, group, others
- Why do we have permissions at all?



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*

Assignment 2

Shell Scripting

Locale

A locale

- Set of parameters that define a user’s cultural preferences
  - Language
  - Country
  - Other area-specific things
- What else does the locale affect?

`locale` command

prints information about the current locale environment to standard output

Environment Variables

- Variables that can be accessed from any child process
- Why do we have these at all? What functions do they serve?

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 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!

Compiled vs. Interpreted

Compiled languages

- Programs are translated from their original source code into machine code that is executed by hardware
- Efficient and fast
- Require recompiling
- Work at low level, dealing with bytes, integers, floating points, etc.
- Ex: C/C++
- When would I want to use a compiled language?

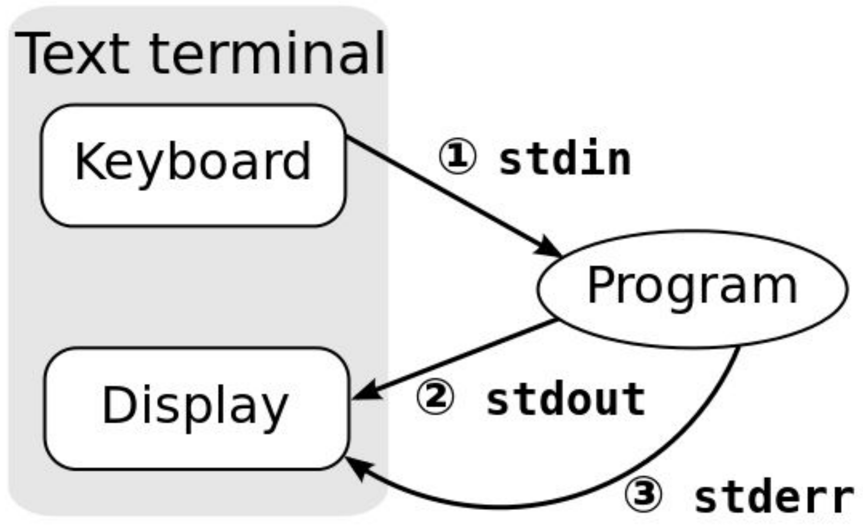
Interpreted languages

- Interpreter program (the shell) reads commands, carries out actions commanded as it goes
- Much slower execution
- Portable
- High-level, easier to learn
- Ex: PHP, Ruby, bash
- When would I want to use an interpreted language?

Why do we have the notion of compiled and interpreted languages?
Why not just have one type of language?

Standard Streams

- Every program has these 3 streams to interact with the world
  - stdin (0): contains data going into a program
  - stdout (1): where a program writes its output data
  - stderr (2): where a program writes its error msgs



Redirection and Pipelines

- *program < file* redirects *file to programs's stdin*:

  ```
  cat <file
  ```
- *program > file* redirects *program*'s stdout to *file2*:

  ```
  cat <file >file2
  ```
- *program 2> file* redirects *program*'s stderr to *file2*:

  ```
  cat <file 2>file2
  ```
- *program >> file* **appends** program's stdout to *file*
- *program1 | program2* assigns stdout of *program1* as the stdin of *program2; text 'flows' through the pipeline*

  ```
  cat <file | sort >file2
  ```

Why would we want to redirect I/O? What are some examples of use cases for I/O redirection? How do we implement this in C?

Regular Expressions

- Notation that lets you search for text with a particular pattern:
  - For example: starts with the letter a, ends with three uppercase letters, etc.
- Why do these exist? Why not just program our own text searching? Are the expressions the same across languages? Platforms?
- What's the difference between a basic and an extended regular expression? When would I use either?
- How do I write a regular expression to accomplish x?

- http://regexpal.com/ to test your regex expressions
- Simple regex tutorial http://www.icewarp.com/support/online_help/203030104.htm

4 Basic Concepts

- Quantification
  - How many times of previous expression?
  - Most common quantifiers: ?(0 or 1), *(0 or more), +(1 or more)
- Grouping
  - Which subset of previous expression?
  - Grouping operator: ()
- Alternation
  - Which choices?
  - Operators: [] and |
    - Hello|World   [A B C]
- Anchors
  - Where?
  - Characters: ^ (beginning) and $ (end)
- How do I use a combination of the above to accomplish tasks?

Regular Expressions

Character	BRE / ERE	Meaning in a pattern
\	Both	Usually, turn off the special meaning of the following character. Occasionally, enable a special meaning for the following character, such as for \(...\) and \{...\}.
.	Both	Match any single character except NUL. Individual programs may also disallow matching newline.
*	Both	Match any number (or none) of the single character that immediately precedes it. For EREs, the preceding character can instead be a regular expression. For example, since . (dot) means any character, **.*** means "match any number of any character." For BREs, * is not special if it's the first character of a regular expression.
^	Both	Match the following regular expression at the beginning of the line or string. BRE: special only at the beginning of a regular expression. ERE: special everywhere.

Regular Expressions (cont'd)

$	Both	Match the preceding regular expression at the end of the line or string. BRE: special only at the end of a regular expression. ERE: special everywhere.
[...]	Both	Termed a bracket expression, this matches any one of the enclosed characters. A hyphen (-) indicates a range of consecutive characters. (Caution: ranges are locale-sensitive, and thus not portable.) A circumflex (^) as the first character in the brackets reverses the sense: it matches any one character not in the list. A hyphen or close bracket (]) as the first character is treated as a member of the list. All other metacharacters are treated as members of the list (i.e., literally). Bracket expressions may contain collating symbols, equivalence classes, and character classes (described shortly).
\{*n*,*m*\}	BRE	Termed an *interval expression*, this matches a range of occurrences of the single character that immediately precedes it. \{*n*\} matches exactly n occurrences, \{*n*,\} matches at least n occurrences, and \{*n*,*m*\} matches any number of occurrences between n and m. n and m must be between 0 and RE_DUP_MAX (minimum value: 255), inclusive.
\(\)	BRE	Save the pattern enclosed between \(and \) in a special *holding space*. Up to nine subpatterns can be saved on a single pattern. The text matched by the subpatterns can be reused later in the same pattern, by the escape sequences \1 to \9. For example, **\(ab\).*\1** matches two occurrences of ab, with any number of characters in between.

Regular Expressions (cont'd)

\n	BRE	Replay the nth subpattern enclosed in \\(and \\) into the pattern at this point. n is a number from 1 to 9, with 1 starting on the left.
{*n*,*m*}	ERE	Just like the BRE \\{*n*,*m*\\} earlier, but without the backslashes in front of the braces.
+	ERE	Match one or more instances of the preceding regular expression.
?	ERE	Match zero or one instances of the preceding regular expression.
|	ERE	Match the regular expression specified before or after.
( )	ERE	Apply a match to the enclosed group of regular expressions.

Matching Multiple Characters with One Expression

*	Match zero or more of the preceding character
{*n*}	Exactly n occurrences of the preceding regular expression
{*n*,}	At least n occurrences of the preceding regular expression
{*n*,*m*}	Between n and m occurrences of the preceding regular expression

Examples

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 (e.g., toltoy, tolstoy, tolWHOtoy, and so on)

Text Processing Tools

- You should be familiar with:
  - `wc`: outputs a one-line report of lines, words, and bytes
  - `head`: extract top of files
  - `tail`: extracts bottom of files
  - `sort`: sort lines of text files
  - `comm`: compare multiple files
  - `tr`: translate or delete characters
  - `grep`: print lines matching a pattern
  - `sed`: filtering and transforming text
- What are the differences between tr, sed, and grep?
- When would I use each one?
- How can I combine and use these tools together?

`wc`, `head`, and `tail`

`wc`: print line, word, and byte counts for each file

- Usage: **`wc`** `[`*`OPTION`*`]... [`*`FILE`*`]...`
  - **-m**, **--chars:** print the number of characters
  - **-w**, **--words:** print the number of words
  - **-l**, **--lines:** print the number of newlines

`head`: output the first part of files

- Defaults to displaying the first 10 lines of each file
- Usage: **`head`** `[OPTION]... [FILE]...`
  - **-n**, **--lines**=*[-]K*: print the first K lines instead of the first 10; with the leading '-', print all but the last K lines of each file

`tail`: output the last part of files

- Defaults to displaying the last 10 lines of each file
- Usage: **`tail`** `[OPTION]... [FILE]...`
  - **-n**, **--lines**=*[-]K*: print the last K lines instead of the last 10; with the leading '-', print all but the first K lines of each file

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

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

- Usage: sort [OPTION]… [FILE]…
  - Passing filename of - will cause sort to read from stdin
  - **u:** unique sort, removes duplicates
  - **r:** reverse sort order
- Sort order depends on locale (C locale: ASCII sorting)

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

- Usage: comm [OPTION]…FILE1 FILE2
  - **1/2/3:** suppresses given column number of output

`tr`: translate **or** delete characters

- Usage: tr [OPTION]…SET1 [SET2]
  - **c:** use the complement of SET1
  - **d:** delete characters in SET1, do not translate

You've implemented a version of `tr`. How did you do that?

`grep` and `sed`

`grep`: print lines matching a pattern

- Usage: `grep [OPTIONS] PATTERN [FILE...]`
  - **-E**: Interpret PATTERN as an extended regular expression (ERE)
    - Defaults to using BRE unless specified otherwise.
  - **-F**: Interpret PATTERN as a list of fixed strings, separated by newlines, any of which is to be matched.
  - **-i**: Ignore case in both the PATTERN and the input files.
  - **-v**: Invert the sense of matching, to select non-matching lines.

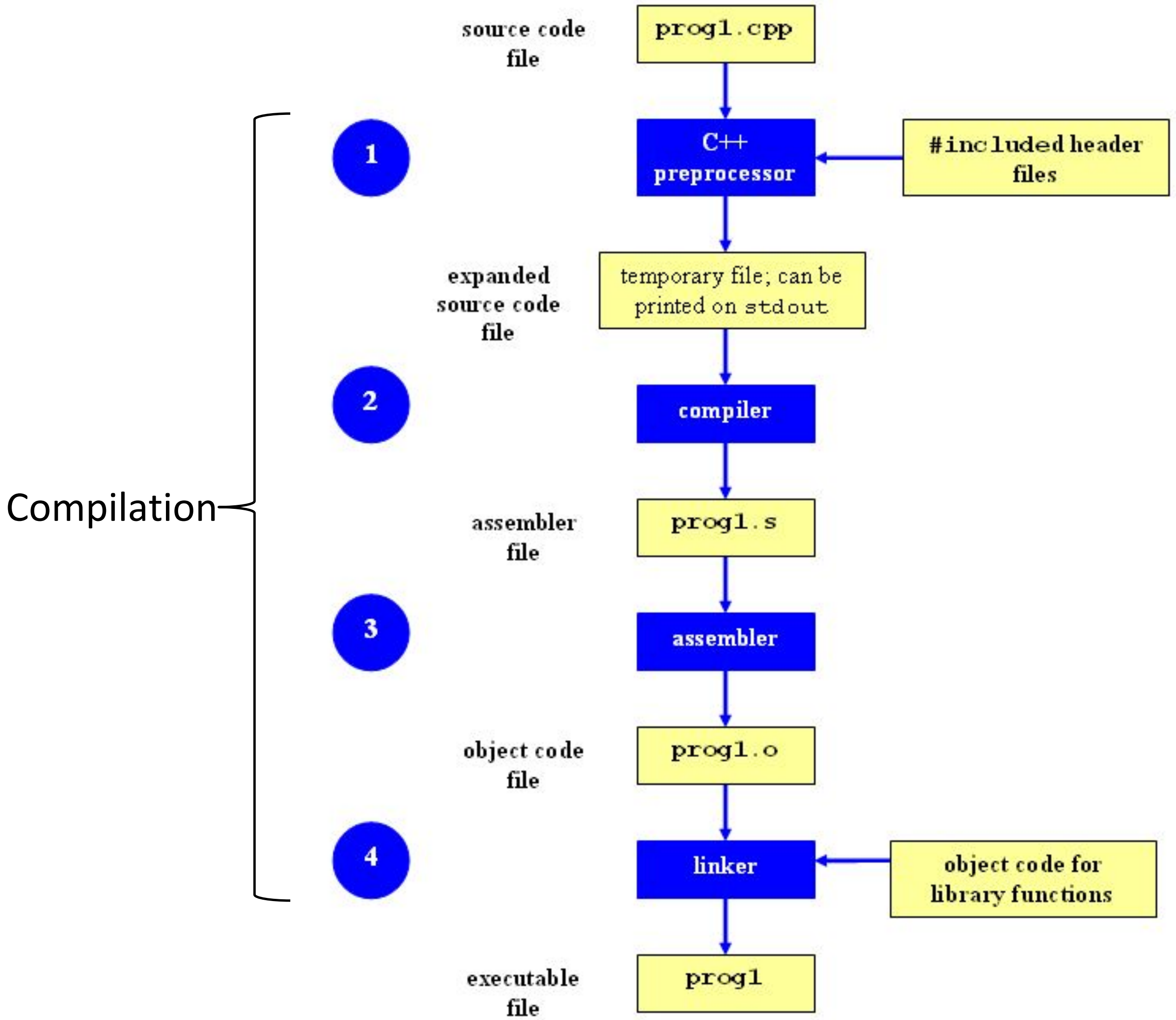
`sed`: stream editor for filtering and transforming text

- Usage: `sed [OPTION]... PATTERN [input-file]...`
  - PATTERN must represent: `‘s/regExpr/replText/’` where
    - `regExpr` is the pattern to be replaced
    - `replText` is the replacement for text matching `regExpr`
  - **-r**: use extended regular expressions in the pattern

Assignment 3

Modifying and rewriting software

Compilation Process



Compilation Process

- Why do we have this process?
- What are the different components of the process?
    - “I just typed gcc to compile my programs… does that mean gcc has all of the components within it?”
- Why can’t I execute individual object code files?
- What are the differences between open source and closed source software? When would I want to use one or the other?

Make

- Utility for managing large software projects
- Compiles files and keeps them up-to-date
- Efficient Compilation (only files that need to be recompiled)
- Why do we have make at all?
  - why don’t we just run ‘gcc …’ from the terminal

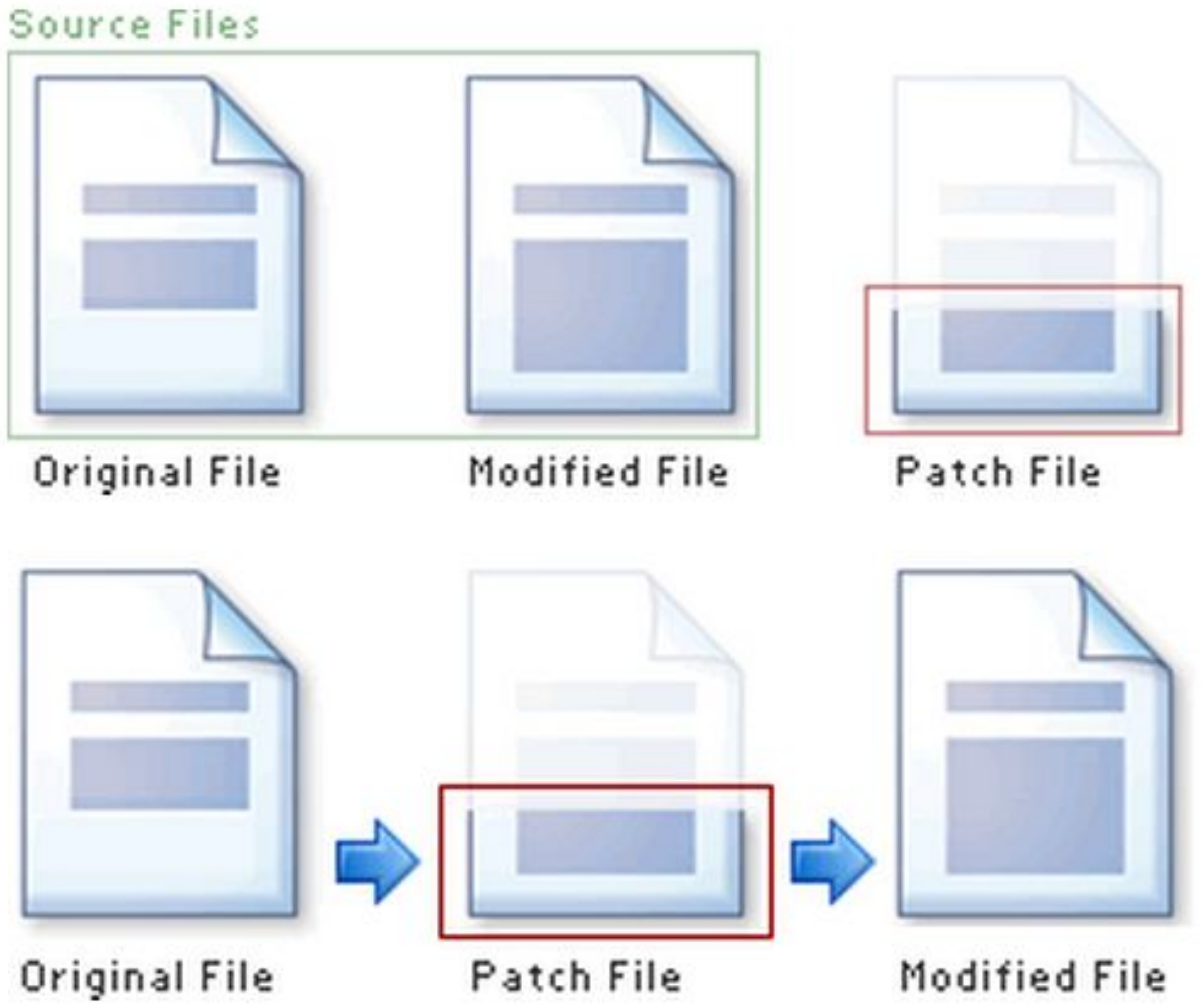
Build Process

- **configure**
  - Script that checks details about the machine before installation
    - Dependency between packages
  - Often creates ‘Makefile’
- **make**
  - Requires ‘Makefile’ to run
  - Compiles all the program code and creates executables in current temporary directory
- **make install**
  - make utility searches for a label named install within the Makefile, and executes only that section of it
  - executables are copied into the final directories (system directories)

Patching

- A patch is a piece of software designed to fix problems with or update a computer program
- It’s a diff file that includes the changes made to a file
- A person who has the original (buggy) file can use the patch command with the diff file to add the changes to their original file
- Why not just change the original source code to fix it? Why do we have patches?

Applying a Patch



diff Unified Format

- diff –u original_file modified_file

- --- path/to/original_file
- +++ path/to/modified_file

- @@ -l,s +l,s @@
  - @@: beginning of a hunk
  - l: beginning line number
  - s: number of lines the change hunk applies to for each file
  - A line with a:
    - - sign was deleted from the original
    - + sign was added to the original
    - stayed the same

What is Python?

- Not just a scripting language
- Object-Oriented language
  - Classes
  - Member functions
- Compiled and interpreted
  - Python code is compiled to bytecode
  - Bytecode interpreted by Python interpreter
- Not as fast as C but easy to learn, read and use
- Why is python powerful? Why is it popular?
- You should know how to write basic python programs

Assignment 4

C Programming and Debugging

C Language

- Subset of C++ (very similar)
- Compiling 'C' only: gcc -std=c99 binsortu.c
- Built-in types:
  - Integers, Floating-point, character strings
  - No bool, false is 0 and true is anything else
- No classes, but we have **structs**
  - No methods and access modifiers

```
struct Song {
    short duration;
    struct Date published;
};
```

Pointers

`double x, y, *ptr;`	Two double variables and a pointer to double.
`ptr = &x;`	Let ptr point to x.
`*ptr = 7.8;`	Assign the value 7.8 to the variable x.
`*ptr *= 2.5;`	Multiply x by 2.5.
`y = *ptr + 0.5;`	Assign y the result of the addition x + 0.5.

Pointers to Functions

```
double (*funcPtr)(double, double);
double result;

// Let funcPtr point to the function pow( ).
// The expression *funcPtr now yields the
// function pow( ).
funcPtr = pow;

// Call the function referenced by funcPtr.
result = (*funcPtr)( 1.5, 2.0 );

// The same function call.
result = funcPtr( 1.5, 2.0 );
```

Dynamic Memory Management

- **malloc(size_t size)**: allocates a block of memory whose size is at least *size*.
- **free(void *ptr)**: frees the block pointed to by ptr
- **realloc(void *ptr, size_t newSize)**: Resizes allocated block

```
Rectangle_t *ptr =
(Rectangle_t*)malloc(sizeof(Rectangle_t));

if(ptr == NULL) {
    printf(“Malloc failed!”);
    exit(-1);
}
else {
    //Perform tasks with the memory
    free(ptr);
    ptr = NULL;
}


ptr = (Rectangle_t*)
    realloc(ptr, 3*sizeof(Rectangle_t))
```

Opening & Closing Files

```
FILE *fopen( const char *
  restrict filename, const char *
  restrict mode );
int fclose( FILE *fp );
```

Common Streams and their file pointers

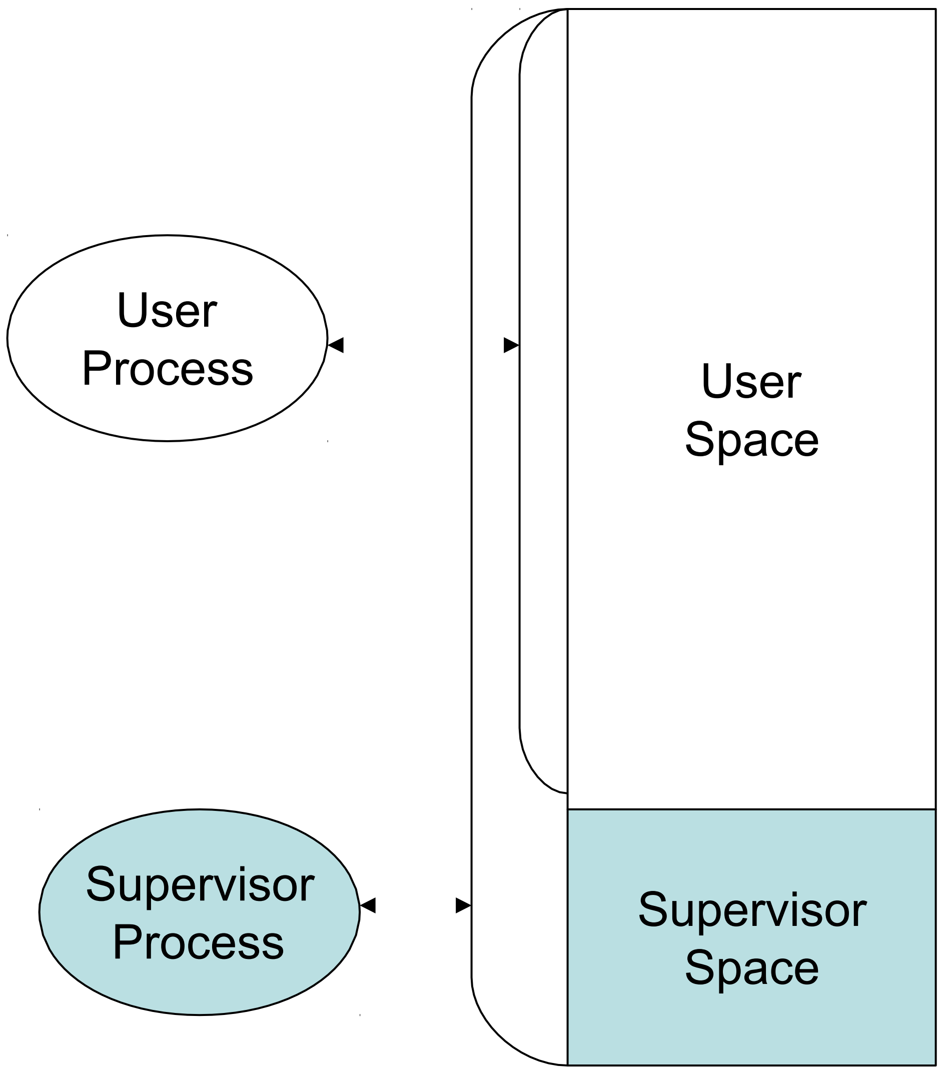
- Standard input: `stdin`
- Standard output: `stdout`
- Standard error: `stderr`

Assignment 5

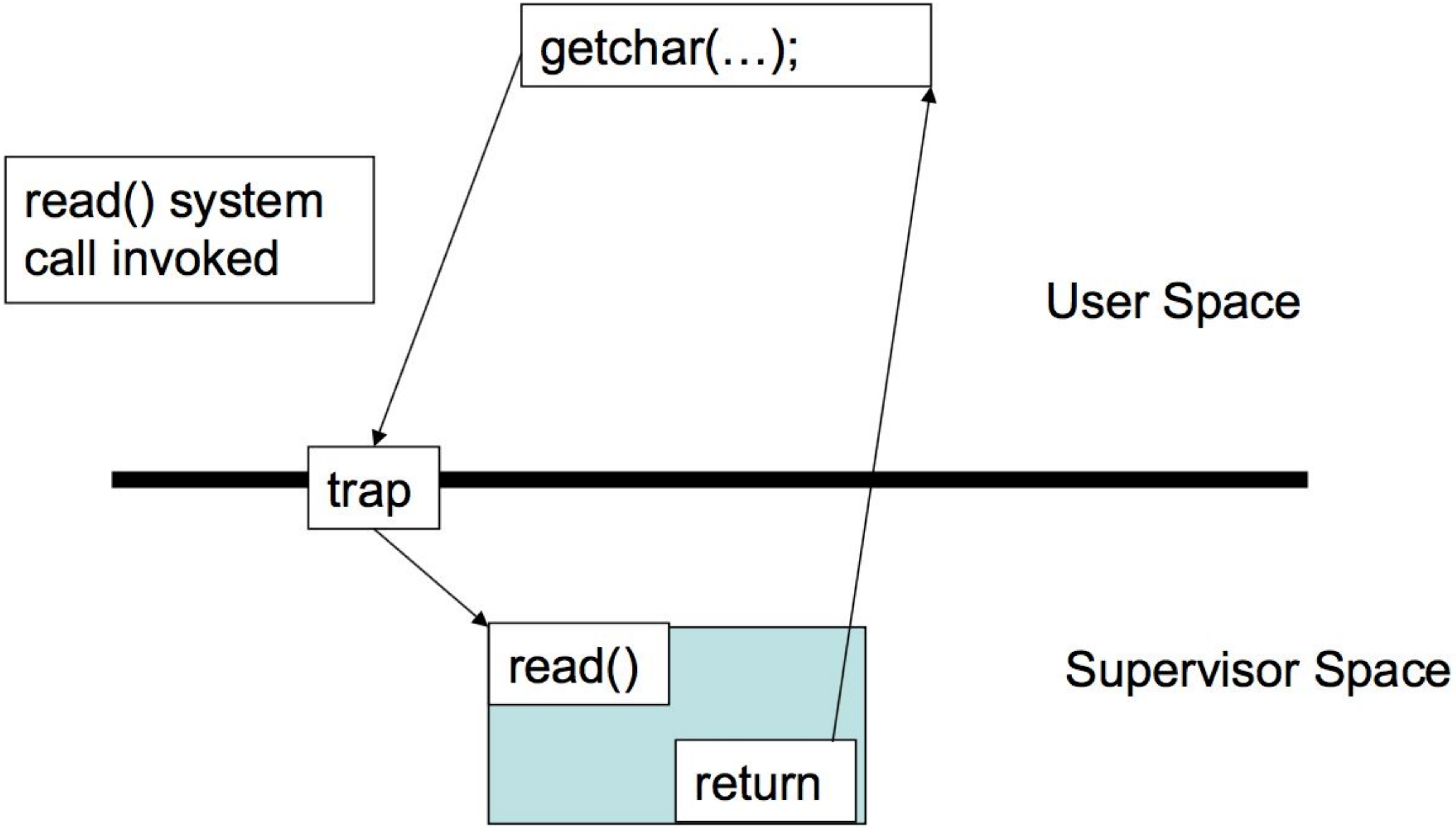
System Call Programming

Processor Modes

- Mode bit may define areas of memory to be used when the processor is in supervisor mode vs user mode



System Calls



Trap: System call causes a switch from user mode to kernel mode

System calls

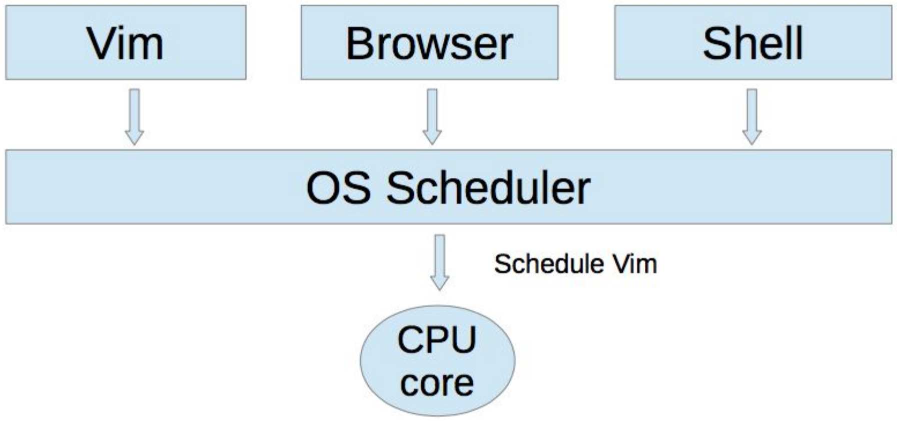
- `ssize_t **read**(int fildes, void *buf, size_t nbyte)`
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to read
- `ssize_t **write**(int fildes,const void *buf,size_t nbyte)`
  - fildes: file descriptor
  - buf: buffer to write to
  - nbyte: number of bytes to write
- `int **open**(const char *pathname,int flags,mode_t mode)`
- `int **close**(int fd)`
- `int **fstat**(int fd, struct stat *buf)`
  - Returns information about the file with the descriptor fd to buf
- File descriptors:
  - 0 stdin, 1 stdout, 2 stderr
- *Why are these system calls and not just regular library functions?*

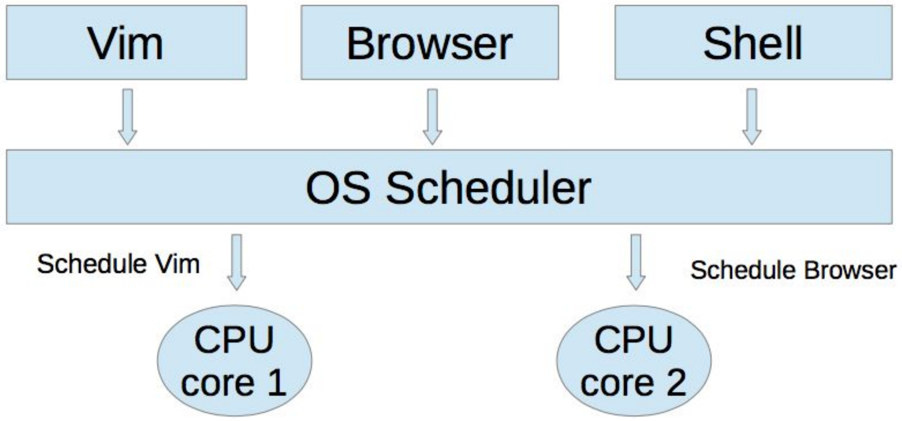
Assignment 6

Multithreading

Multitasking

- Run multiple processes **simultaneously** to increase performance
- Processes do not share internal structures (stacks,globals,etc)
  - Communicate via **IPC** (inter-process communication) methods
    - Pipes, sockets, signals, message queues
- **Single core: Illusion** of parallelism by switching processes quickly (**time-sharing**). Why is illusion good?
- **Multi-core: True** parallelism. Multiple processes execute **concurrently** on different CPU cores





Multithreading properties

- Efficient way to **parallelize** tasks
- **Thread switches are less expensive** compared to process switches (context switching)
- Inter-thread communication is easy, via **shared global** data
- Need **synchronization** among threads accessing same data

Pthread API

```
#include <pthread.h>
```

- `int` **`pthread_create`**`(pthread_t *thread,`
  `const pthread_attr_t *attr,void*`
  `(*thread_function) (void*), void *arg);`
  - Returns 0 on success, otherwise returns non-zero number

- `void` **`pthread_exit`**`(void *retval);`

- `int` **`pthread_join`**`(pthread_t thread, void **retval);`
  - Returns 0 on success, otherwise returns non zero error number

Thread synchronization

- **Mutex (mutual exclusion)**
  - Thread 1
    - Mutex.lock()
      - Read balance
      - Deduct 50 from balance
      - Update balance with new value
    - Mutex.unlock()
  - Thread 2
    - Mutex.lock()
      - Read balance
      - Add 150 to balance
      - Update balance with new value
    - Mutex.unlock()
  - balance = 1100
- Only one thread will get the mutex. Other thread will **block in Mutex.lock()**
- Other thread can start execution only when the thread that holds the mutex calls **Mutex.unlock()**

Questions

- What can go wrong in multithreading?
  - What are race conditions?
  - What is deadlock?
- What are some approaches to make multithreading safer?
  - What are the possible advantages or disadvantages of each of these approaches?
- Comment: It may be useful to consider changing an algorithm to make it safer . . .

Assignment 7

SSH

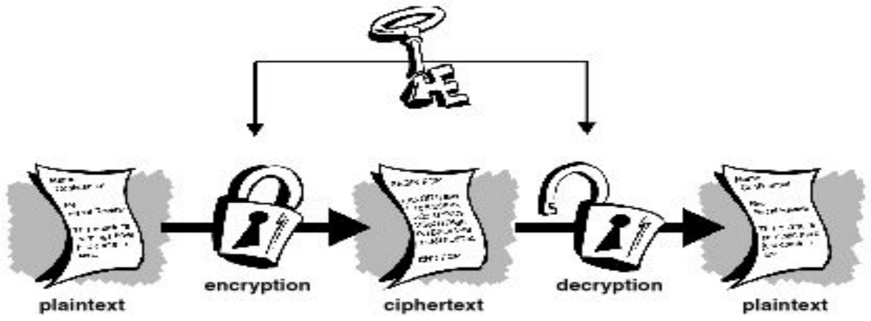
Communication Over the Internet

What type of guarantees do we want?

- **Confidentiality**
  - Message secrecy: “Can anybody else read this message?”
- **Data integrity**
  - Message consistency: “Has someone altered this message?”
- **Authentication**
  - Identity confirmation: “Is this message really from Alice?”
- **Authorization**
  - Predicated permission: “Can bad people get into my system?”

Cryptography

- Plaintext - actual message
- Ciphertext - encrypted message (unreadable gibberish)
- Encryption - converting from plaintext to ciphertext
- Decryption - converting from ciphertext to plaintext
- Secret key
  – part of the mathematical function used to encrypt\decrypt
  – Good key makes it hard to get back plaintext from ciphertext



Symmetric-key Encrption

- Same secret key used for encryption and decryption
- **Example** : Data Encryption Standard (**DES**)
- **Caesar's cipher**
  - Map the alphabet to a shifted version
    - ABCDEFGHIJKLMNOPQRSTUVWXYZ
    - DEFGHIJKLMNOPQRSTUVWXYZABC
  - Plaintext – SECRET. Ciphertext – VHFUHW
  - Key is 3 (number of shifts of the alphabet)
- **Key distribution** is a problem
  - The secret key has to be delivered in a safe way to the recipient
  - Chance of key being compromised

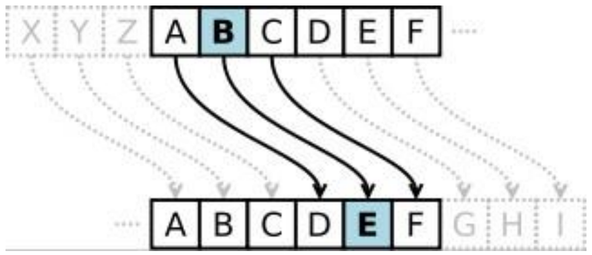


Image Source: wikipedia

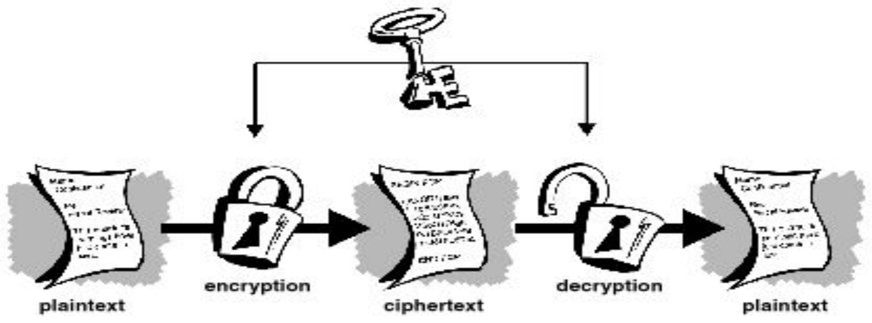


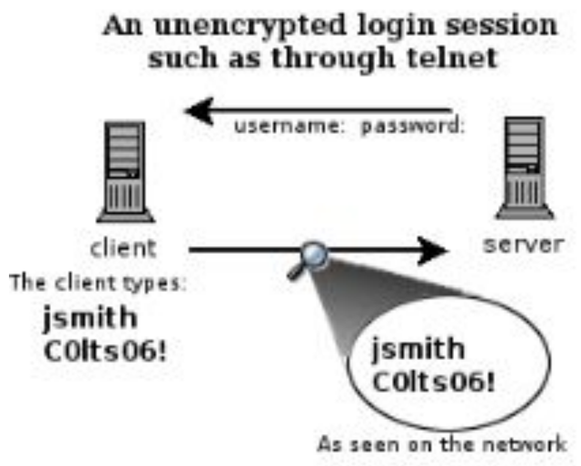
Image Source: gpgtools.org

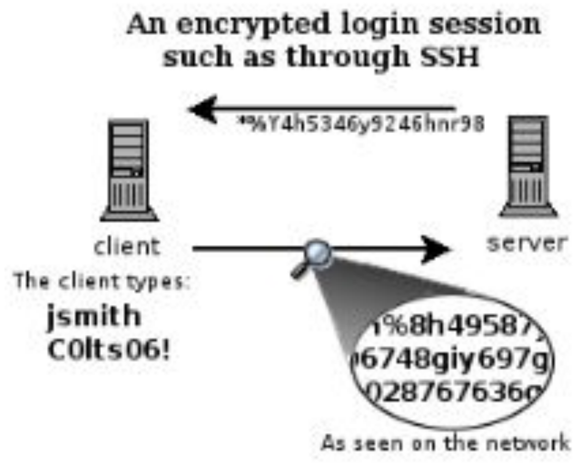
Public-Key Encryption (Asymmetric)

- Uses a pair of keys for encryption
  - Public Key - published and well known to everyone
  - Private - Secret key known only to the owner
- Encryption
  - Use public key to encrypt messages
  - Anyone can encrypt message, but they cannot decrypt the ciphertext
- Decryption
  - Use private key to decrypt messages
- In what scheme is this encryption useful?

Secure Shell (SSH)

- Telnet
  - Remote access
  - Not encrypted
  - Packet sniffers can intercept sensitive information (username/password)
- SSH
  - run processes remotely
  - encrypted session
  - Session key (secret key) used for encryption during the session





Session Encryption

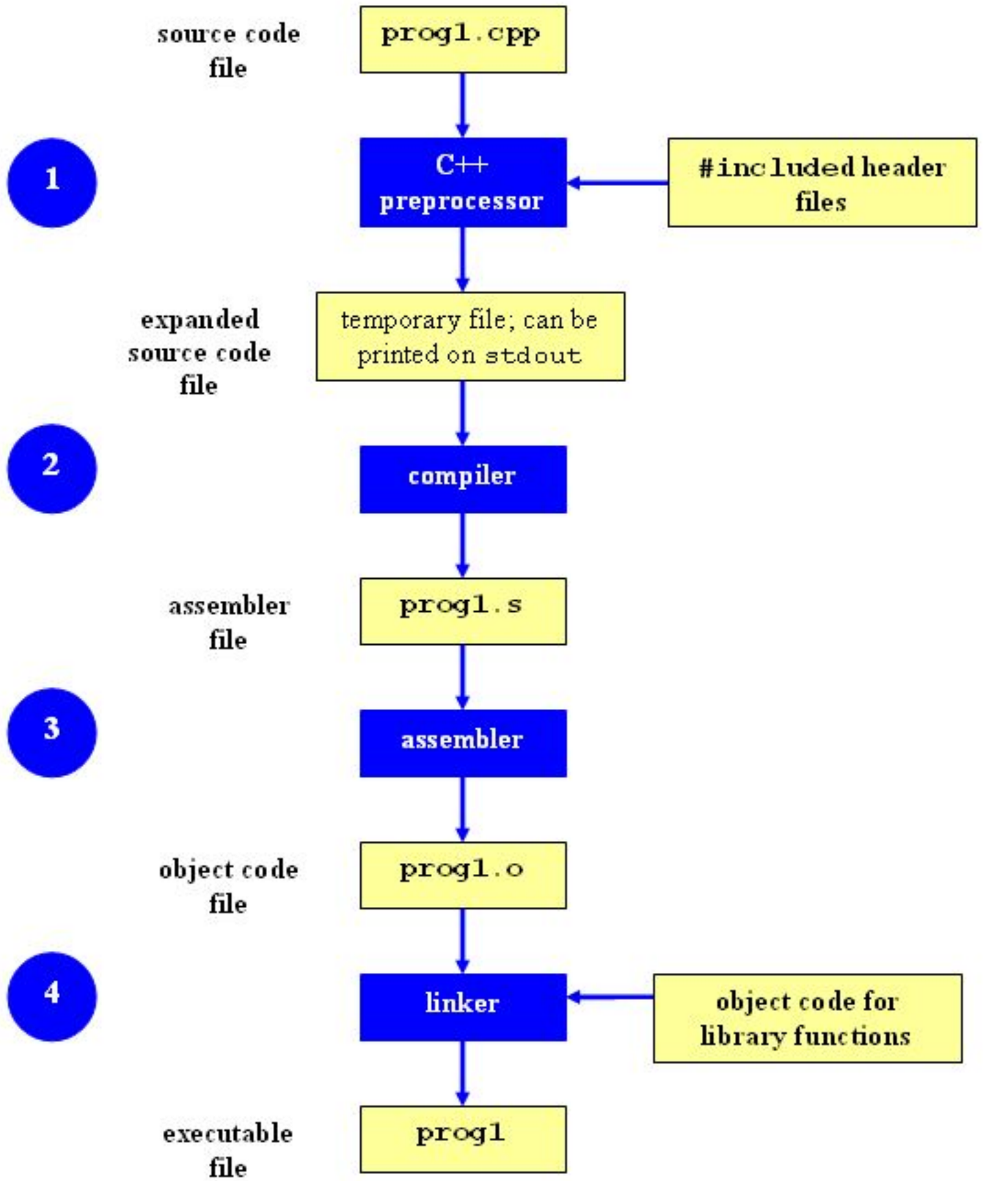
- Client and server agree on a **symmetric encryption key** (session key)
- All messages sent between client and server
  - encrypted at the sender with session key
  - decrypted at the receiver with session key
- anybody who doesn't know the session key (hopefully, no one but client and server) doesn't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

Assignment 8

Dynamic Linking

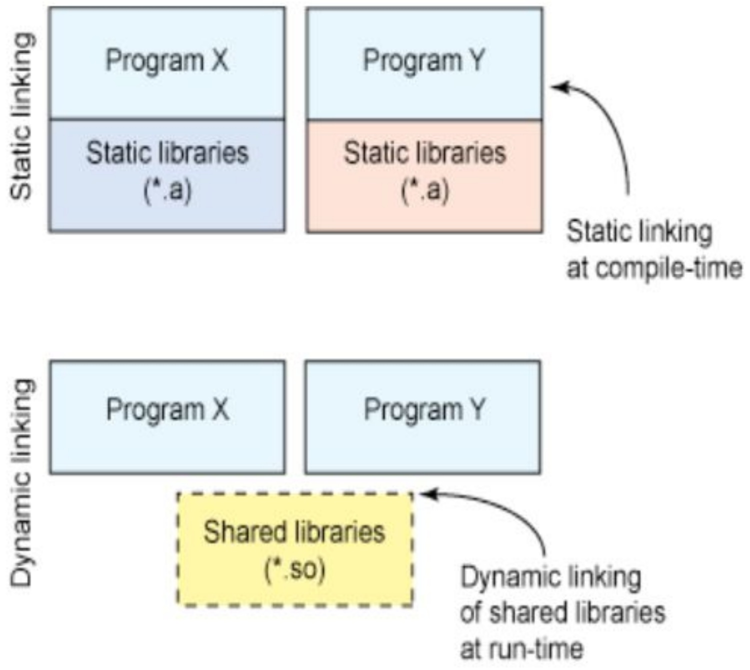
Compilation Process

- Preprocessor
  - Expand Header includes, macros, etc
  - -E option in gcc to show the resulting code
- Compiler
  - Generates machine code for certain architecture
- Linker
  - Linkall modules together
  - Address resolution
- Loader
  - Loads the executable to memory to start execution



Linux Libraries

- **Static Library**
  - Statically linked
  - Every program has its own copy
  - More space in memory
  - Tied to a specific version of the lib. New version of the lib requires recompile of source code.
- **Shared Library (binding at run-time)**
  - Dynamically loaded/linking
    - **Dynamic Linking** – The OS loads the library when needed. A dynamic linker does the linking for the symbol used.
    - **Dynamic Loading** – The program "actively" loads the library it needs (DL API – dlopen(), dlclose()). More control to the program at run-time. Permits extension of programs to have new functionality.
  - Library is shared by multiple programs
  - Lower memory footprint
  - New version of the lib does not require a recompile of source code using the lib



Img Source : http://www.ibm.com/developerworks/library/l-dynamic-libraries/

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
  - .dll (dynamically linked library) on Windows

How are libraries dynamically linked?

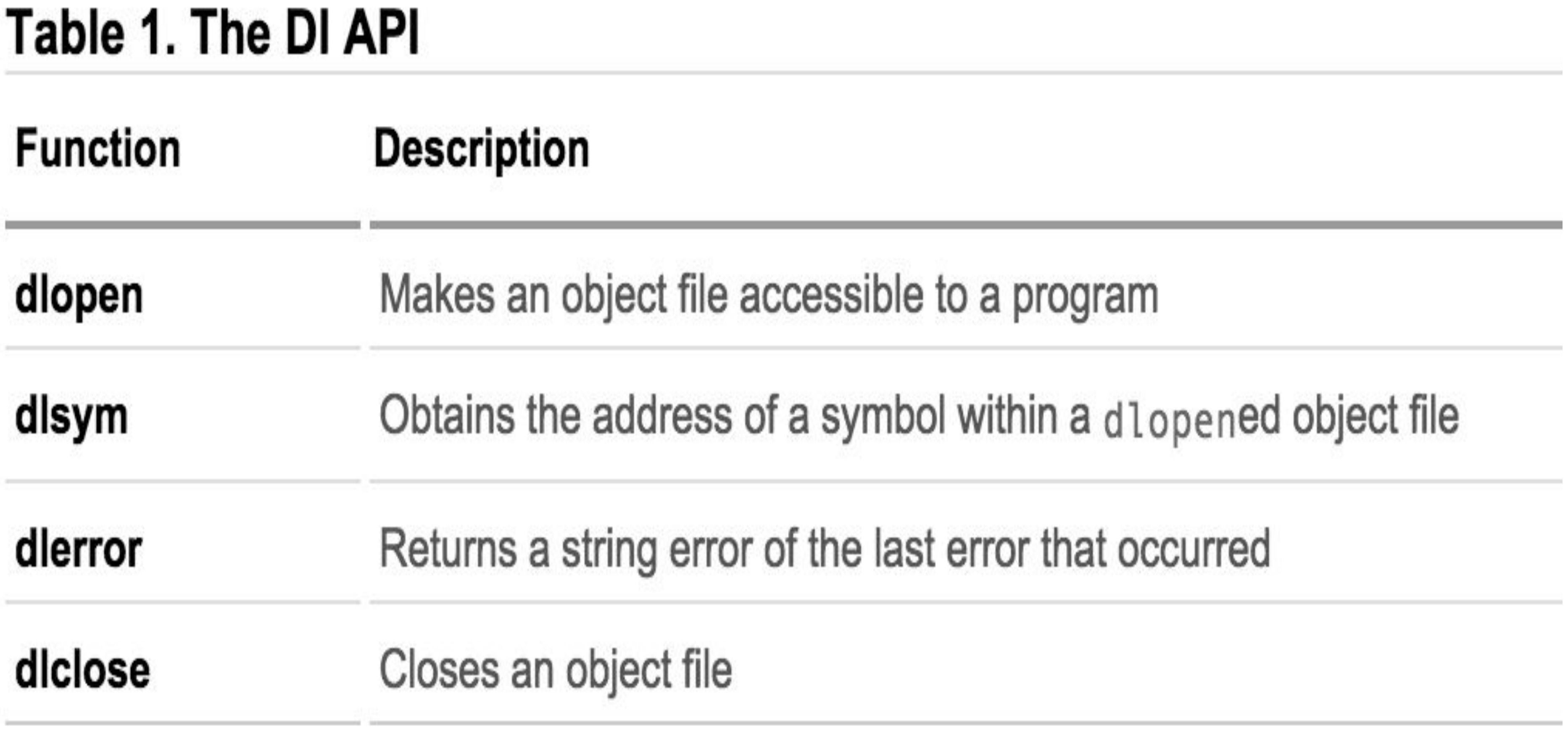


Table 1. The Dl API

Function	Description
dlopen	Makes an object file accessible to a program
dlsym	Obtains the address of a symbol within a `dlopen`ed object file
dlerror	Returns a string error of the last error that occurred
dlclose	Closes an object file

Questions

What are the advantages and disadvantages of dynamic linking (as opposed to static linking)?

How does the build process change with dynamic linking?

What, if any, computational or data overhead could dynamic linking require?

Assignment 9

Change Management

Source/Version Control

- Track changes to code and other files related to the software
  - What new files were added? What
  - changes made to files?
  - Which version had what changes?
  - Which user made the changes?
- Track entire history of the software
- Version control software
  - GIT, Subversion,
    Perforce

This seems complicated. Why bother with source control?
What are the strengths and weaknesses of source control?
When would I want to use it? How do I use it?

Terms Used

- **Repository**
  - Files and folder related to the software code
  - Full History of the software
- **Working copy**
  - Copy of 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

Terms used

- **Head**
  - Refers to a commit object
  - There can be many heads in a repository
- **HEAD**
  - Refers to the currently active head
- **Detached HEAD**
  - If a commit is not pointed to by a branch
  - This is okay if you want to just take a look at the code and if you don't commit any new changes
  - If the new commits have to be preserved then a new branch has to be created
    - `git checkout v3.0 -b BranchVersion3.1`
- **Branch**
  - Refers to a head and its entire set of ancestor commits
- **Master**
  - Default branch

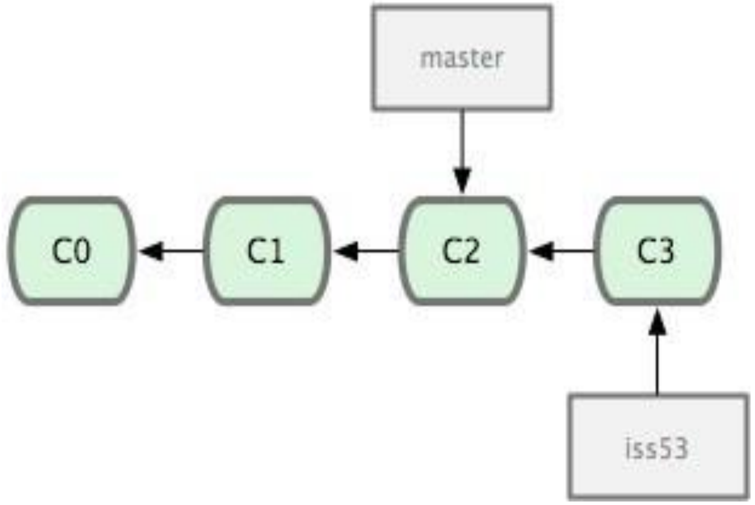
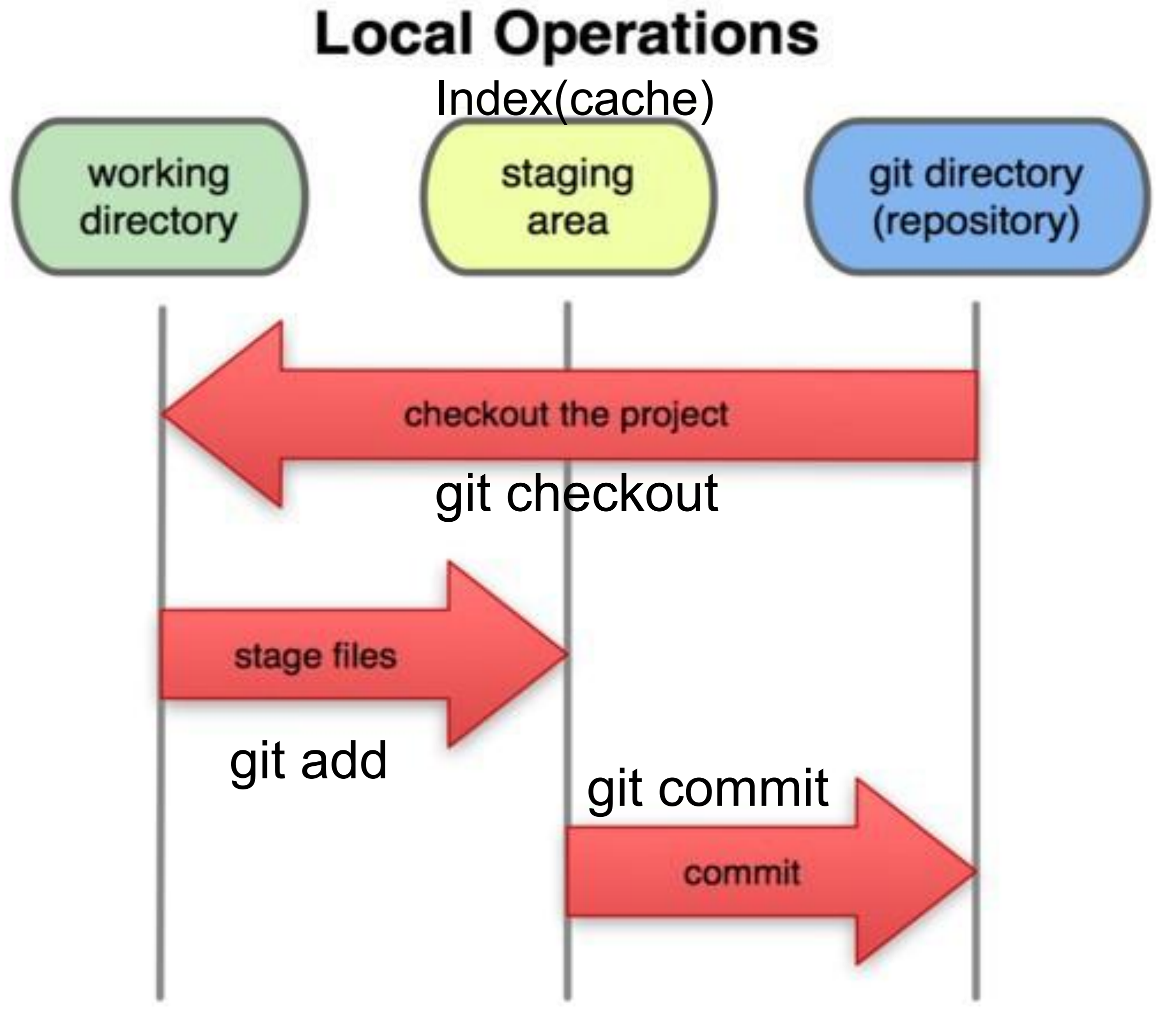


Image Source: git-scm.com

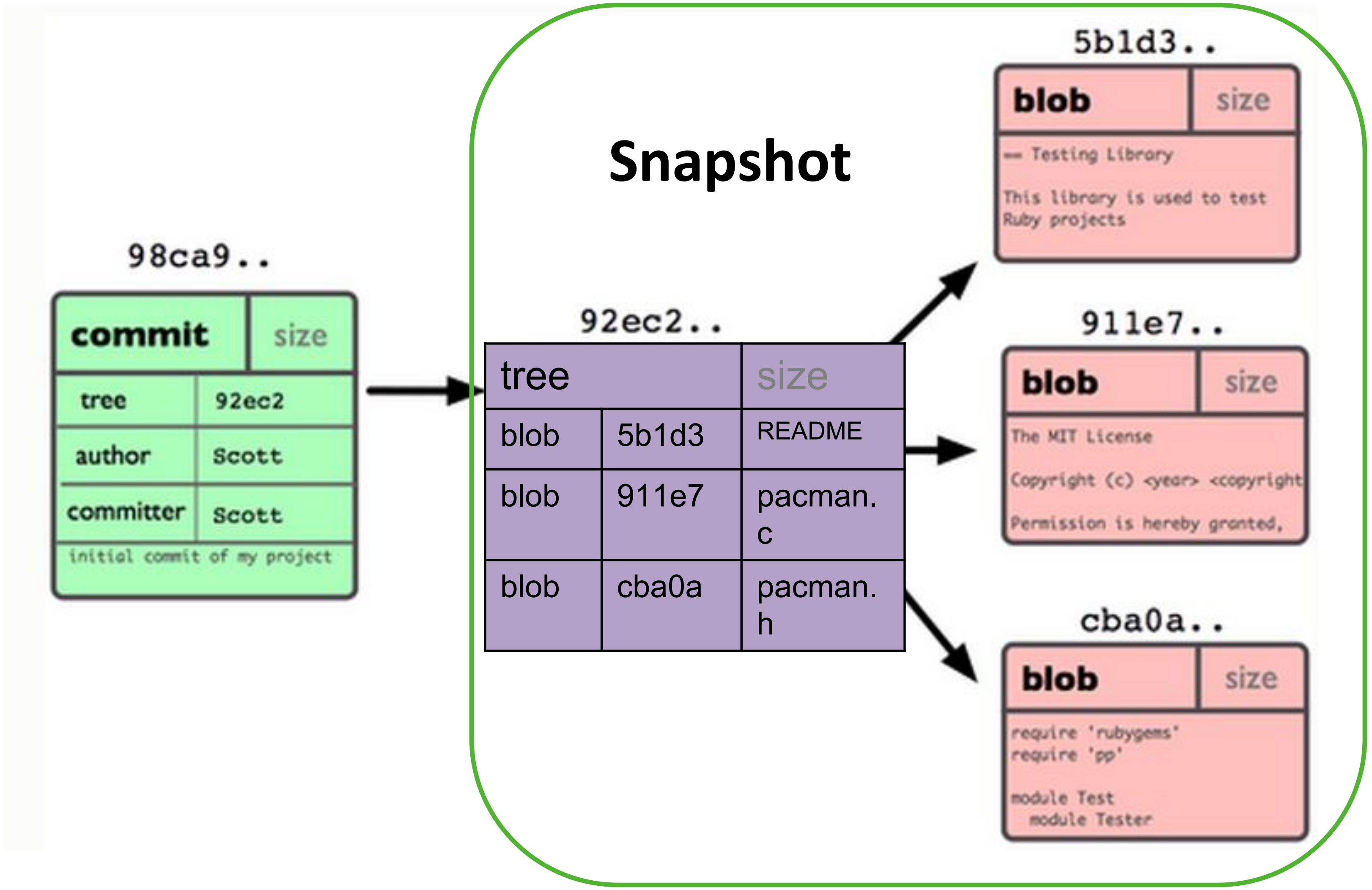
What Is a Branch?

- A pointer to one of the commits in the repo (head) + all ancestor commits
- When you first create a repo, are there any branches?
  - Default branch named ‘master’
- The default master branch
  - points to last commit made
  - moves forward automatically, every time you commit

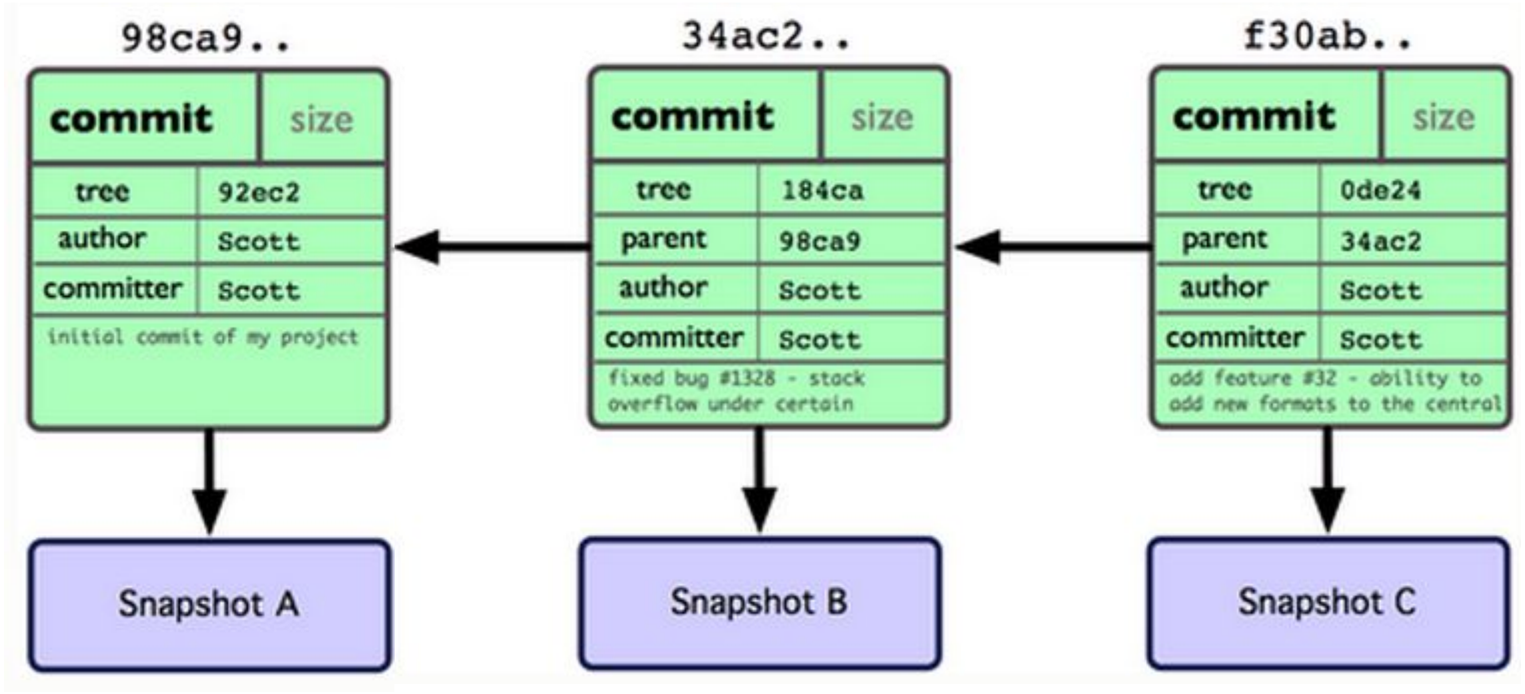
Git States


Local Operations
Index(cache)
working directory
staging area
git directory (repository)
checkout the project
git checkout
stage files
git add
git commit
commit

Git Repo Structure



After 2 More Commits…



Git commands

- Repository creation
  - $ git init (Start a new repository)
  - $ git clone (Create a copy of an exisiting repository)
- Branching
  - $ git checkout <tag/commit> -b <new_branch_name> (creates a new branch)
- Commits
  - $ git add (Stage modified/new files)
  - $ git commit (check-in the changes to the repository)
- Getting info
  - $ git status (Shows modified files, new files, etc)
  - $ git diff (compares working copy with staged files)
  - $ git log (Shows history of commits)
  - $ git show (Show a certain object in the repository)
- Getting help
  - $ git help

You should be familiar with how these commands work and when to use them.

More Git Commands

- Reverting
  - $ git checkout HEAD main.cpp
    - Gets the HEAD revision for the working copy
  - $ git checkout -- main.cpp
    - Reverts changes in the working directory
  - $ git revert
    - Reverting commits (this creates new commits)
- Cleaning up untracked files
  - $ git clean
- Tagging
  - Human readable pointers to specific commits
  - $ git tag -a v1.0 –m 'Version 1.0'
    - This will name the HEAD commit as v1.0

You should be familiar with how these commands work and when to use them.

Questions

- What is the difference between a working copy and the repository?
- What is a commit? What should be in a commit? How many files should commits contain?
- Why bother having branches at all? Why can’t we just all work on the same single master branch?
- What happens when we perform a merge? How does it work?