CSE 15L: Software Tools and Techniques Laboratory

Winter 2021 - http://ieng6.ucsd.edu/~cs15x

Instructors: Gary Gillespie Keith Muller

Class sessions will be recorded and made available to students asynchronously.

Schedule

Final Exam Review!

Today:

1. Standard IO vs Pipes
2. Managing Permissions with chmod
3. **`git`** Review

Final Exam Format

- Exam will be on Canvas (canvas.ucsd.edu)
- Available to start any time on either assigned day
  - Select any CSE 15L exam (Monday or Friday)
  - Start exam from 12:01 am to 11:59 pm (Pacific Time)
  - Once started, you have 90 min to complete that exam
  - Can't select another exam once you've started an exam
- Questions will be multiple-choice, fill-in-the-blank, matching, and short answer (including writing short functions in Bash)
- Open book, open note, open Internet but no discussing the exam with other students until all finals have ended
- You may use a Bash terminal while taking the exam, but it is not required for any question

Standard IO vs Pipes

Standard IO

- Each shell (and all programs) usually have three “files” open when they start up
  - Standard input (stdin), keyboard, FD 0

    ```
    $ command < somefile
    ```

  - Standard output (stdout), display, FD 1

    ```
    $ command > afile1
    $ command 1> afile2
    ```

  - Standard error (stderr), error (display) FD 2

    ```
    $ command 2> afile3
    ```

- To redirect stderr to be the same file as stdout

  ```
  $ command > afile4 2>&1
  ```

Standard IO

- Be cautious of reading/writing to the same file!
  Writing with `>` will erase it prior to reading from stdin

```
$ uniq < shakespeare.txt > shakespeare.txt
```

- Append to a file using `>>`

```
$ command >> afile1
$ command 2>> afile3
```

- Direct input (terminal or script) to stdin using

  `<<  HERE` or `<<-  HERE` (ignores leading tabs)

  `<<<` (directs input from the command line)

Pipes

Instead of a program writing data to a file on disk and the next reading it, pass data via a memory buffer.

In terms of output to terminal:

```
$ ls -l > flist.txt; grep -e "-rwxrw----"
                flist.txt
```

is functionally (not performance) equivalent to

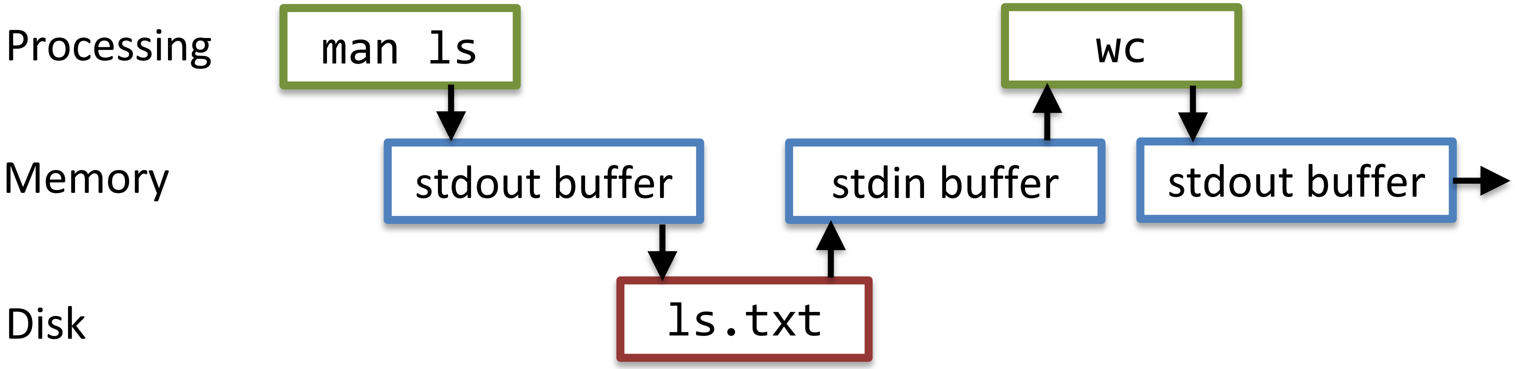
```
$ ls -l | grep -e "-rwxrw----"
```

Difference in resulting file system and speed!

1. First command creates a file (flist.txt) in the local directory
2. Writing to/reading from a file is slow compared to accessing memory

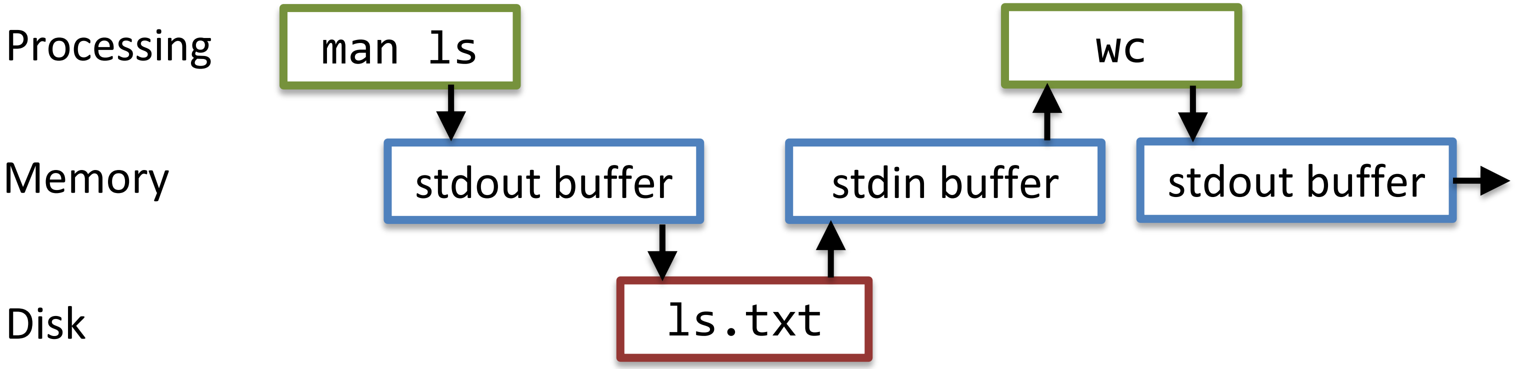
Standard IO vs Piping

```
man ls > ls.txt; wc < ls.txt
```



Standard IO vs Piping

```
man ls > ls.txt; wc < ls.txt
```



Processing

Memory

Disk

man ls

wc

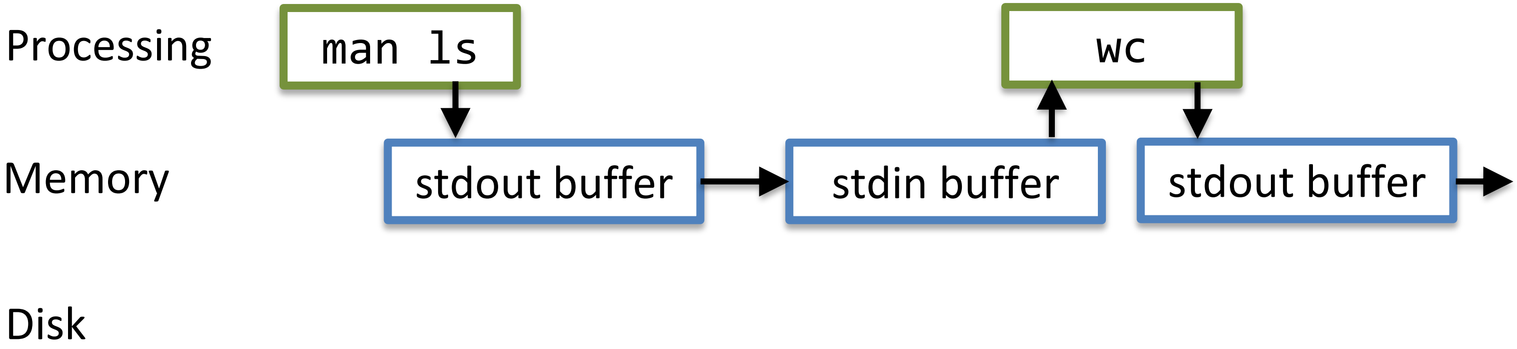
stdout buffer

stdin buffer

stdout buffer

ls.txt

```
man ls | wc
```



Processing

Memory

Disk

man ls

wc

stdout buffer

stdin buffer

stdout buffer

Practice Problem 1

```
$ grep -i "juliet" shakespeare.txt > names.log
$ grep -i "romeo" shakespeare.txt >> names.log
$ sort < names.log > names_sort.log
$ uniq < names_sort.log > names_uniq.log
$ wc –l < names_uniq.log
```

1. (short answer)

a) Explain what this sequence of commands in a script will output

b) how to interpret the output

c) what files (if any) will be added to the file system.

Solution 1

```
$ grep -i "juliet" shakespeare.txt > names.log
$ grep -i "romeo" shakespeare.txt >> names.log
$ sort < names.log > names_sort.log
$ uniq < names_sort.log > names_uniq.log
$ wc –l < names_uniq.log
```

1. Script will output a single value, the number of unique lines in the file shakespeare.txt, containing either the word Juliet or Romeo (case insensitive).

   The files: `names.log`, `names_sort.log`, and `names_uniq.log` will be added to the local directory if they did not exist and over-written

Practice Problem 2

```
$ grep -i "juliet" shakespeare.txt > names.log
$ grep -i "romeo" shakespeare.txt >> names.log
$ sort < names.log > names_sort.log
$ uniq < names_sort.log > names_uniq.log
$ wc –l < names_uniq.log
```

Knowing that the `–e {pattern}` option can be used multiple times with `grep` to find more than 1 pattern

(e.g., `grep –e "thing1" –e "thing2"`)

2. (short answer) how can you re-write the above code in a single line using `;` and/or pipes so that the output to the terminal is the same and no additional files are created?

Solution 2

```
$ grep -i "juliet" shakespeare.txt > names.log
$ grep -i "romeo" shakespeare.txt >> names.log
$ sort < names.log > names_sort.log
$ uniq < names_sort.log > names_uniq.log
$ wc -l < names_uniq.log
```

2. The above code can be rewritten as follows:

```
$ grep -i -e "juliet" -e "romeo"shakespeare.txt |
sort | uniq | wc -l
```

Managing Permissions with `chmod`

User Categories and Permissions

- chmod (change mode) modifies the permissions of nodes in the file system (like files, directories, etc.)

- Three types of Permissions:

  read (r), write (w), and execute (x)

- Categories:

  owner (u), group (g), others (o), and all (a)

Interpreting Permissions

- Output of `ls -l` will display permissions for each category:

```
$ ls -l
-rwxrw-r-- 1 mosterta staff 17 Nov 6 12:01 temp.txt
```

`-rwxrw-r--`

directory owner group others

- In bitfield, d r w x represents 1 and – represents 0
  - Read Permission (0b100 = 4)
  - Write Permission (0b010 = 2)
  - Execute Permission (0b001 = 1)

chmod

- Incremental permission change using add (+) or remove (-)

```
$ chmod ug+x myscript.sh
$ chmod o-rwx privatefile.txt
```

- Setting permission using =

```
$ chmod g=rw shareddoc.txt
```

- Perform multiple, separate by comma (,)

```
$ chmod og=r,u=rwx testscript.sh
```

Multiple changes must be passed in a single argument! **No spaces!**

Octal Syntax

- Set permissions using sum of binary permission values!

Read (0b100 = 4), Write (0b010 = 2), Execute (0b001 = 1)

Read+Write = 6 (0b110)

Read+Write+Execute = 7 (0b111)

```
$ chmod ### filename
```

owner
group
others

Practice Problem 3

(Fill in the blank) Assume the file `error.log` originally has access permissions of "`-r--r--rwx`" access permissions.

3. What chmod command using octal notation can be used to set the file permissions to an equivalent state as after executing the following commands?

```
$ chmod a+w,o-x error.log
$ chmod go-r error.log
```

Equivalent command:

```
chmod (fill) (fill)
```

Solution 3

(Fill in the blank) Assume the file `error.log` originally has access permissions of "`-r--r--rwx`" access permissions.

3. What chmod command using <u>octal notation</u> can be used to set the file permissions to an equivalent state as after executing the following commands?

```
$ chmod a+w,o-x error.log        "-rw-rw-rw-"
$ chmod go-r error.log           "-rw--w--w-"
```

r (0b100 = 4) **+ w** (0b010 = 2) **= rw** (0b110 = 6)

Equivalent command:

```
chmod 622 error.log
```

Practice Problem 4

(Fill in the blanks)

4. Given the following sequence fill in the output of `ls -l`.

```
$ ls -l
-rwxrwxr-- 1 mosterta staff 17 Nov 6 12:01 test.txt
$ cp -p test.txt temp.txt (note: -p  retains file permissions)
$ chmod 560 test.txt
$ chmod go-rx temp.txt
$ mkdir solns
$ mv temp.txt solns.txt
$ ls -l
drwxr-xr-x 2 mosterta staff 4096 Nov 6 12:03 ___a___
___b___    1 mosterta staff 17 Nov 6 12:03 ___c___
___d___    1 mosterta staff 17 Nov 6 12:02 test.txt
```

Solution 4

(Fill in the blanks)

4. Given the following sequence fill in the the output of `ls -l`.

```
$ ls -l
-rwxrwxr-- 1 mosterta staff 17 Nov 6 12:01 test.txt
$ cp -p test.txt temp.txt (note: -p retains file permissions)
$ chmod 560 test.txt       test.txt: -r-xrw----
$ chmod go-rx temp.txt     temp.txt: -rwx-w----
$ mkdir solns              new dir: solns: d?????????
$ mv temp.txt solns.txt    temp.txt -> solns.txt
$ ls -l
drwxr-xr-x 2 mosterta staff 4096 Nov 6 12:03 solns
-rwx-w---- 1 mosterta staff 17 Nov 6 12:03 solns.txt
-r-xrw---- 1 mosterta staff 17 Nov 6 12:02 test.txt
```

Aside: umask

permissions assigned to newly created files or directories are modified by the **umask** value

% umask - display current umask
% umask xyz - sets new umask to an octal value **xyz**
permissions on a newly created file or directory:

1. start with a “default” of 777 for a directory or 666 for a file
2. for each 1 in the binary representation of the umask, change the corresponding bit to 0 in the binary representation of the default

umask is a **reverse mask**:

binary representation specifies which permission bits in the 777 or 666 default will be 0 in the newly created file or directory

aside: umask (files)

if umask is 022	
binary umask representation:	0b000010010 = 022
default file permissions 666:	0b110110110
permissions on a new file:	0b110100100 = 644
if umask is 002	
binary umask representation:	0b000000010 = 002
default file permissions 666:	0b110110110
permissions on a new file:	0b110110100 = 664
if umask is 003	
binary umask representation:	0b000000011 = 003
default file permissions 666:	0b110110110
permissions on a new file:	0b110110100 = 664

?

aside:umask (directories)

if umask is 022

binary umask representation:	0b000010010 = 022
default dir permissions 777:	0b111111111
permissions on new dir:	0b111101101 = 755

if umask is 002

binary umask representation:	0b000000010 = 002
default dir permissions 777:	0b111111111
permissions on new dir:	0b111111101 = 775

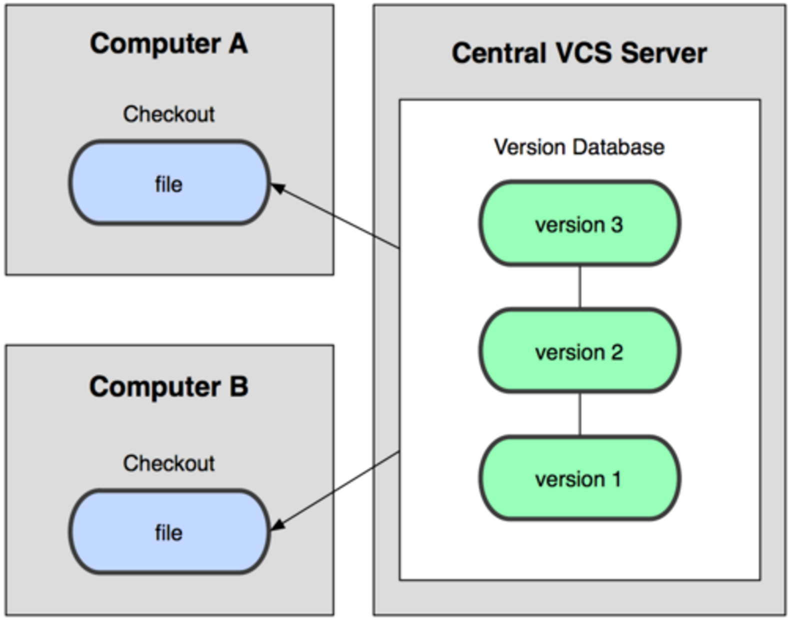
if umask is 003

binary umask representation:	0b000000011 = 003
default dir permissions 777:	0b111111111
permissions on new dir:	0b111111100 = 774

`git` in more detail

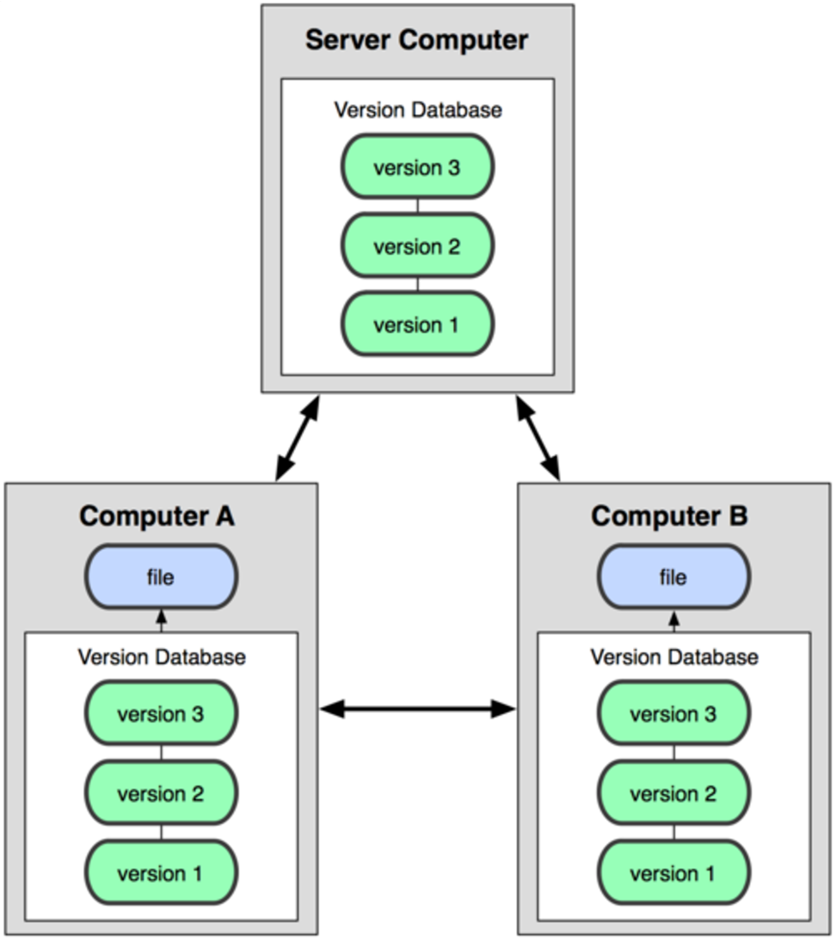
`git` uses a Distributed Model

Centralized Model



(CVS, Subversion, Perforce)

Distributed Model



(Git, Mercurial)
Result: Many operations are local

Getting a Local Repository

- Since git is distributed, a repository will live in your file system in your highest-level project folder!
- The repository is stored in a `.git` folder

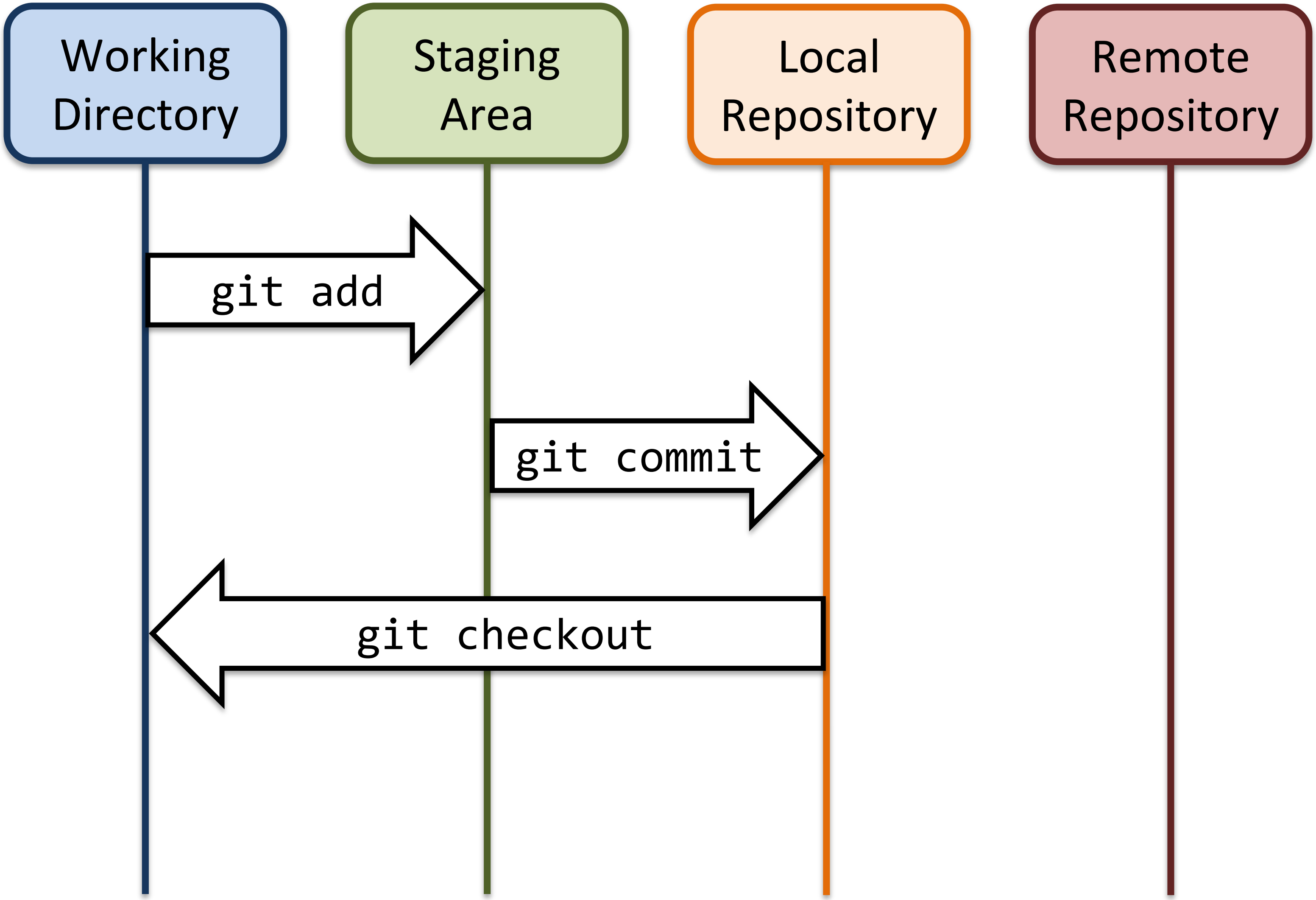
- Create using `git init`, which creates a `.git` folder

  or

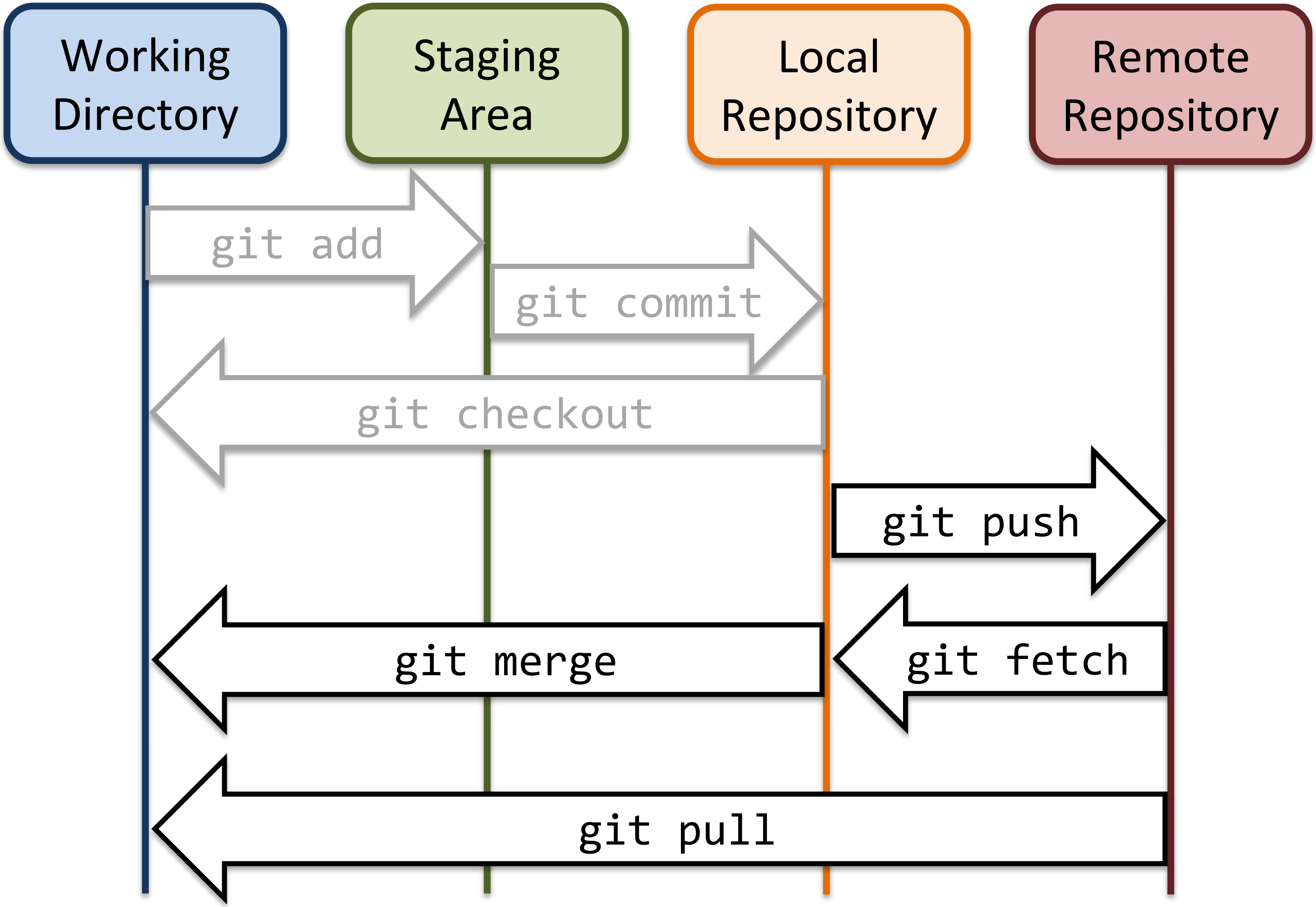
- Clone an existing repo using `git clone <url>`

Without a local repository, you cannot `add`, `pull`, `push`, `fetch`, `checkout`, etc.!

Working with Repos



Working with Repos



File Status in `git`

- `git` tracks file statuses in the project directory
- Files can be:
  - `untracked`: not currently part of the repository
  - `unmodified`: tracked but not changed since the last commit
  - `modified`: tracked and changed since the last commit
  - `staged`: tracked, changed, and added as part of the next commit

Practice Problem 5

(Matching) Match the git commands with their related definitions:

a. `git status` b. `git diff` c. `git log` d. `git hist`
e. `git fetch` f. `git pull` g. `git get` h. `git checkout`

1. (fill)	Command to update your local repo from the remote repo without merging
2. (fill)	Command to get a list of previous commit ids, authors, dates, and messages.
3. (fill)	Command to replace a file in your working directory with the version in the local repo
4. (fill)	Command to list which files are untracked, modified, or staged.

Solution 5

(Matching) Match the git commands with their related definitions:

a. `git status` b. `git diff` c. `git log` d. `git hist`
e. `git fetch` f. `git pull` g. `git get` h. `git checkout`

1. e	Command to update your local repo from the remote repo without merging
2. (fill)	Command to get a list of previous commit ids, authors, dates, and messages.
3. (fill)	Command to replace a file in your working directory with the version in the local repo
4. (fill)	Command to list which files are untracked, modified, or staged.

Solution 5

(Matching) Match the git commands with their related definitions:

a. `git status` b. `git diff` c. `git log` d. `git hist`

e. `git fetch` f. `git pull` g. `git get` h. `git checkout`

1. e	Command to update your local repo from the remote repo without merging
2. c	Command to get a list of previous commit ids, authors, dates, and messages.
3. (fill)	Command to replace a file in your working directory with the version in the local repo
4. (fill)	Command to list which files are untracked, modified, or staged.

Solution 5

(Matching) Match the git commands with their related definitions:

a. `git status` b. `git diff` c. `git log` d. `git hist`

e. `git fetch` f. `git pull` g. `git get` h. `git checkout`

1. e	Command to update your local repo from the remote repo without merging
2. c	Command to get a list of previous commit ids, authors, dates, and messages.
3. h	Command to replace a file in your working directory with the version in the local repo
4. (fill)	Command to list which files are untracked, modified, or staged.

Solution 5

(Matching) Match the git commands with their related definitions:

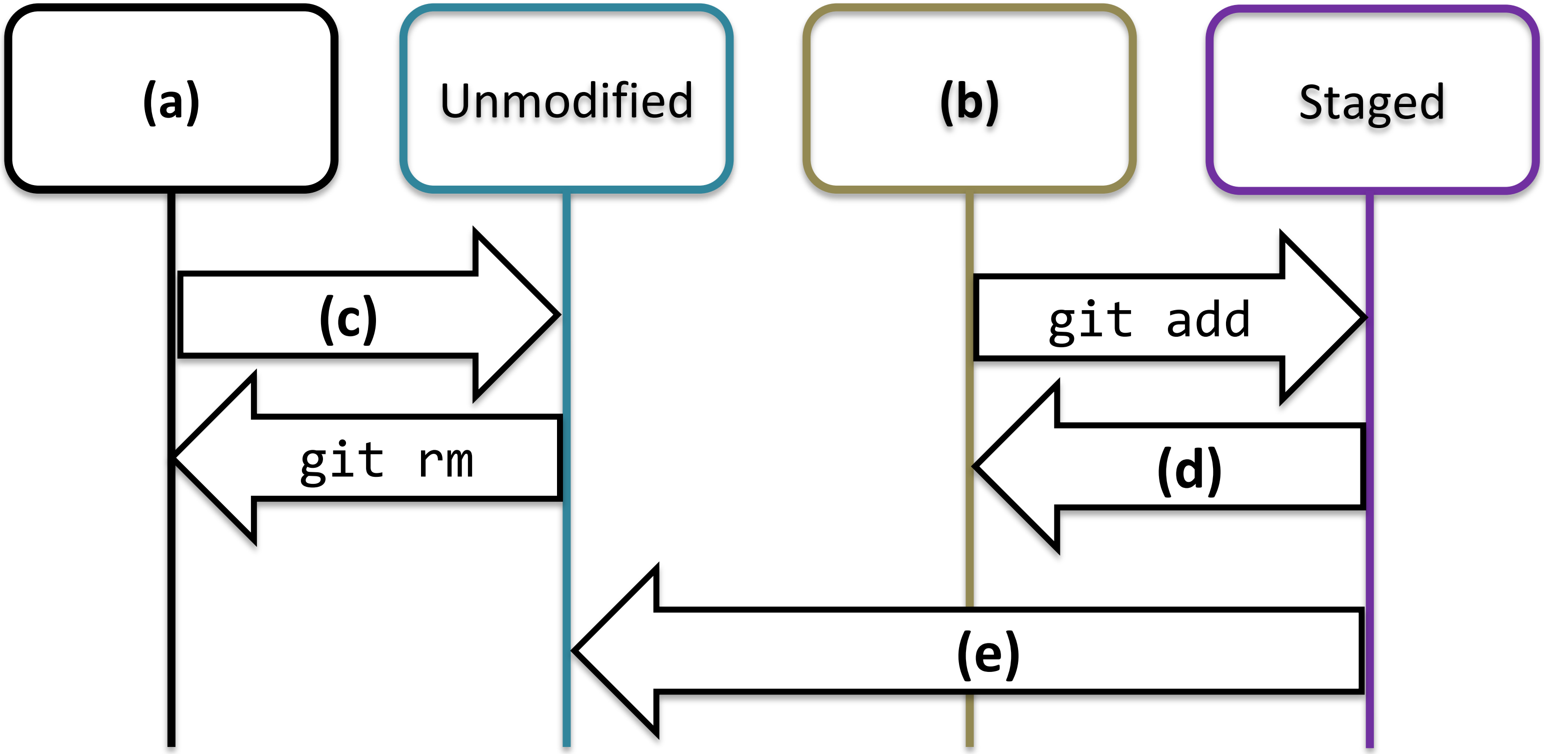
a. `git status` b. `git diff` c. `git log` d. `git hist`

e. `git fetch` f. `git pull` g. `git get` h. `git checkout`

1. e	Command to update your local repo from the remote repo without merging
2. c	Command to get a list of previous commit ids, authors, dates, and messages.
3. h	Command to replace a file in your working directory with the version in the local repo
4. a	Command to list which files are untracked, modified, or staged.

Practice Problem 6

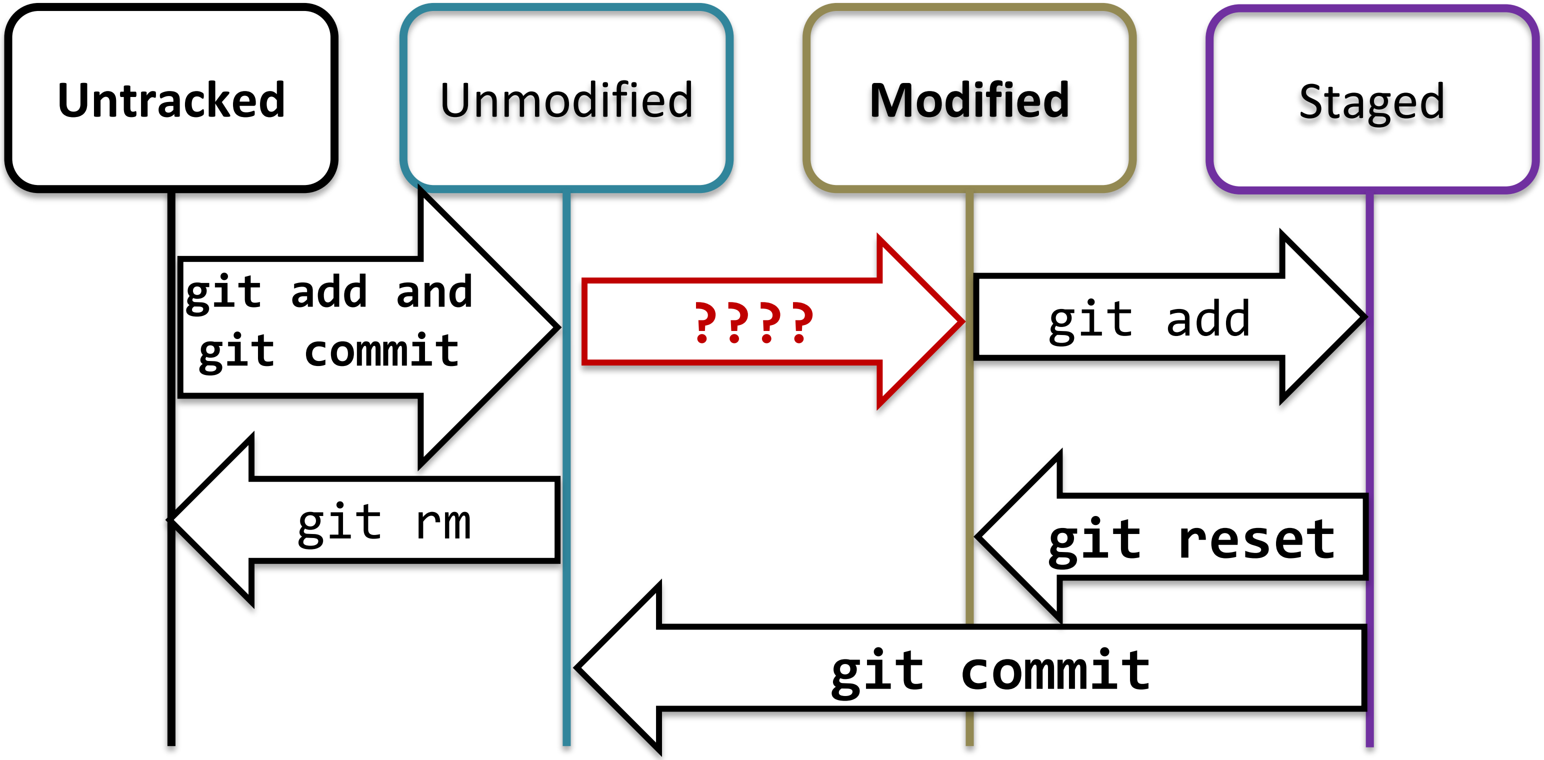
(Fill in) Fill in the file statuses and git command(s) to transition between file statuses.



```
init   clone   branch   checkout   reset   add   commit
pull   fetch   merge   push   status   diff   log   tag
```

Solution 6

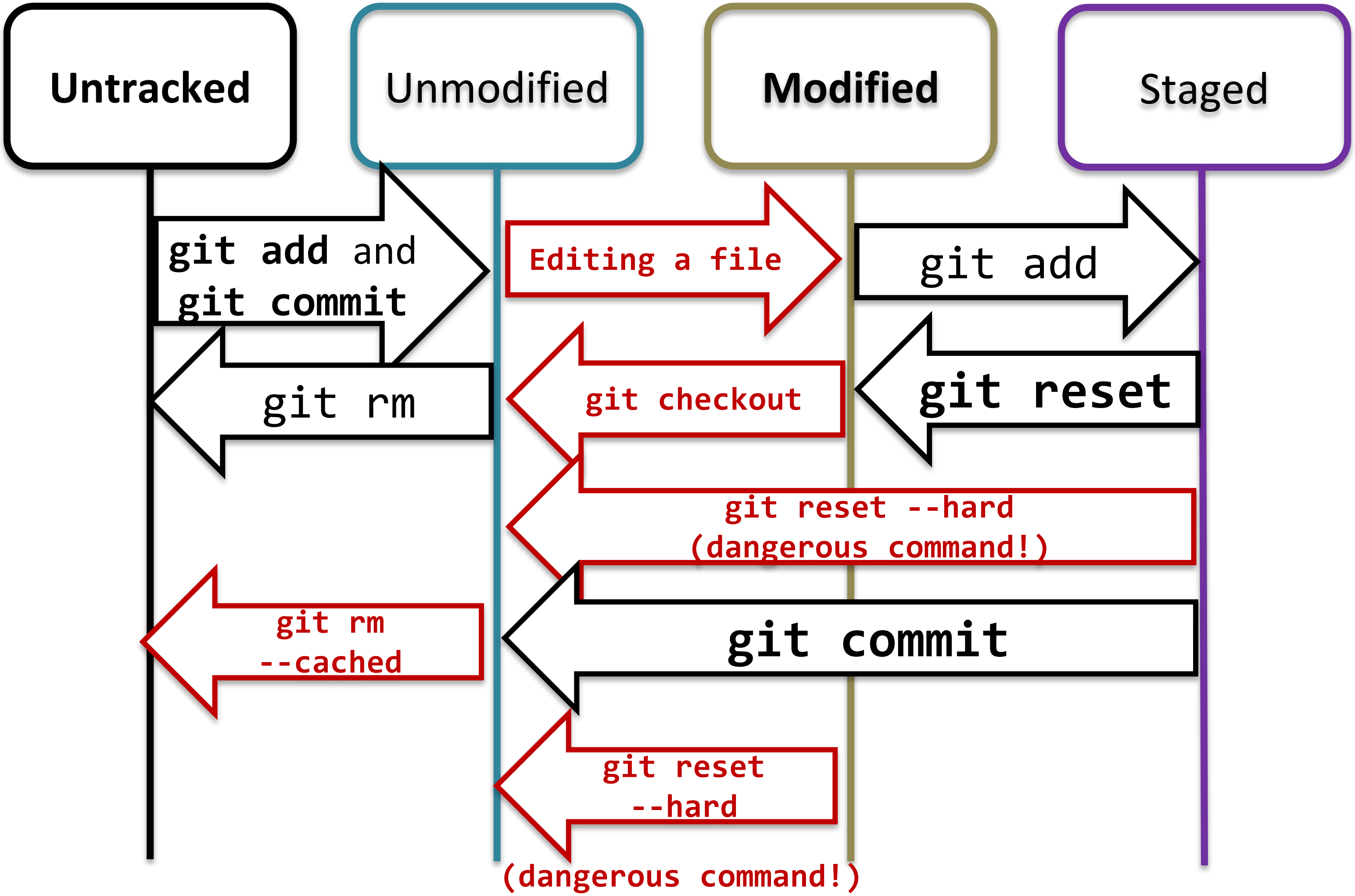
(Fill in) the file statuses and git commands to transition between file statuses.



```
init   clone   branch   checkout   reset   add   commit
pull   fetch   merge   push   status   diff   log   tag
```

Solution 6

(Fill in) the file statuses and git commands to transition between file statuses.



init clone branch checkout reset add commit
pull fetch merge push status diff log tag

Next Lecture

Final Review Round 2