Lecture 3

Tuesday September 15, 2020

Recap

Last time:

- Shell customization
- I/O
- Bash scripting

This time:

- Job Management
- Environment variables
- Version control

Job Control

Job Control

- The shell allows you to manage jobs:
  - Place jobs in the background
  - Move a job to the foreground
  - Suspend a job
  - Kill a job
- Putting an `&` after a command on the command line will run the job in the background
- Why do this?
  - You don’t want to wait for the job to complete
  - You can type in a new command right away
  - You can have a bunch of jobs running at once
- e.g. `./program > output &`

Job Control: `nohup` and Terminal Multiplexers

- Use `nohup` if the job will run longer than your session
  - `nohup ./program &> output &`
- `Terminal multiplexers` are *great* for this
  - 6 Best terminal multiplexers as of 2020
  - Screen
  - tmux

Listing Jobs

The shell assigns a number to each job

```
(base) SEAS-:2020-CS107 $ iacs launch &
[1] 69144
```

The `jobs` command lists all background jobs

```
(base) SEAS-:2020-CS107 $ jobs
[1]  + running    iacs launch
```

kill the foreground job using `Ctrl-C`

Kill a background job using the `kill` command

```
(base) SEAS-:2020-CS107 $ kill %1
(base) SEAS-:2020-CS107 $
[1]  + terminated  iacs launch
```

Breakout Room: Practice Listing Jobs

1. Use the `sleep` command to suspend the terminal session for 60 seconds
   a. Hint: If you're never met `sleep` before, type `man sleep` at the command line
2. Suspend the job using `Ctrl-Z`
3. Now list the jobs using the `jobs` command
4. The job isn't in the background; it's just suspended. Send the job to the background with `bg %n` where `n` is the job id that you obtained from the `jobs` command
5. Bring the `sleep` command (your job) back to the foreground with `fg %n`
6. You could let the job finish (since it's only sleeping for 60 seconds after all), or you can kill it. Up to you.

Environment Variables

Environment Variables

- Unix shells maintain a list of environment variables that have a unique name and value associated with them
  - Some of these parameters determine the behavior of the shell
  - They also determine which programs get run when commands are entered
  - They can provide information about the execution environment to programs
- We can access these variables
  - Set new values to customize the shell
  - Find out the value to accomplish a task
- Use `env` to view environment variables
- Use `echo` to print variables
  - `echo $PWD`
  - The `$` is needed to access the value of the variable

```
(base) SEAS-:2020-CS107 $ env | grep PWD
PWD=/Users/dsondak/Teaching/Harvard/CS107/2020-CS107
OLDPWD=/Users/dsondak/Teaching/Harvard/CS107
```

PATH

- Each time you provide the shell a command to execute, it does the following:
  - Checks to see if the command is a built-in shell command
  - If it's not a built-in shell command, the shell tries to find a program whose name matches the desired command
- How does the shell know where to look on the filesystem?
- The `PATH` variable tells the shell where to search for programs

```
(base) SEAS-:2020-CS107 $ echo $PATH
/Users/dsondak/opt/anaconda3/bin:/Users/dsondak/opt/anaconda3/condabin:/usr/local/lib/ruby/gems/2.7.0/bin:/Users/dsondak/.jenv/shims:/Users/dsondak/.jenv/bin:/opt/
local/bin:/opt/local/sbin:/Users/dsondak/gems/bin:/Users/dsondak/.gem/ruby/2.6.0/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin
```

- The `PATH` is a list of directories delimited by colons
  - It defines a list and search order
  - Directories specified earlier in PATH take precedence
  - The search terminates once the matching command is found
- Add more search directories to your path using export:
  - `export PATH="$PATH:/Users/dsondak"`

Setting Environment Variables

- Setting a Unix environment in bash uses the `export` command
  - `export USE_CUDA=OFF`
- Environment variables that you set interactively are only available in your current shell
  - These settings will be lost if you spawn a new shell
  - To make more lasting changes, alter the login scripts that affect your particular shell (in bash this is `.bashrc`, in zsh this is `.zshrc`)
- An environment variable can be deleted with the `unset` command
  - `unset USE_CUDA`

Version Control

What is version control?

Version control is a way of tracking the change history of a project.

You've probably already engaged in manual version control 😬.

If you finished HW1, then you've definitely started playing around with Git.

Perhaps you have a lot of questions now...

Version Control

- Minimum guidelines: Actually using version control is the first step
- Ideal usage:
  - Put **everything** under version control
  - Consider putting parts of your home directory under version control
    - e.g. `.zshrc`
  - Use a consistent project structure and naming convention
  - Commit often and in logical chunks
  - Write meaningful commit messages
  - Do all file operations in the version control system
    - e.g. `git mv <name> <new_name>`
  - Set up change notifications if working with multiple people

Source Control and Versioning

Why bother?

- Codes evolve over time
  - Sometimes bugs creep in (by you or others)
  - Sometimes the old way was right
  - Sometimes it's nice to look back at the evolution

Version control is a non-negotiable component of any project.

Why???

code	code3.cpp	code_FINAL_new.cpp	code_final_send	code_orig.cpp
code.cpp	code_110303.cpp	code_USE.cpp	code_fix.cpp	code_orig_1.cpp
code1.cpp	code_FINAL.cpp	code_bug_fixes.cpp	code_for_john	code_running.cpp
code2.cpp	code_FINAL_1.cpp	code_bugs.cpp	code_new.cpp	code_send

Reproducibility Maintainability **Project longevity**

So why use version control?

If you ask 10 people, you'll get 10 different answers.

One of the commonalities is that most people don't realize how integral it is to the development process until they've started using it

Some reasons for using version control

- **You can undo anything**
  - git provides a *complete history* of every change that has ever been made to your project
  - timestamped, commented, and attributed
  - If something breaks, you can always go back to a previous state
- **You won't *need* to keep undoing things**
  - You can avoid the massive rollbacks associated with constantly tinkering with a single code
  - This is because proper use of git entails keeping new changes separate from a stable base
- **You can identify exactly when and where changes were made**
  - git allows you to pinpoint when a particular piece of code was changed
  - Finding pieces of code a bug might affect is easy
  - Figuring out why a certain expression was added is also easy
- **Git forces teams to face conflicts directly**
  - It may seem troublesome to deal with conflicts
  - However, this is much better than the alternative - not even knowing there is a conflict

Examples of Version Control

- Mercurial
- Git

} Distributed Version Control

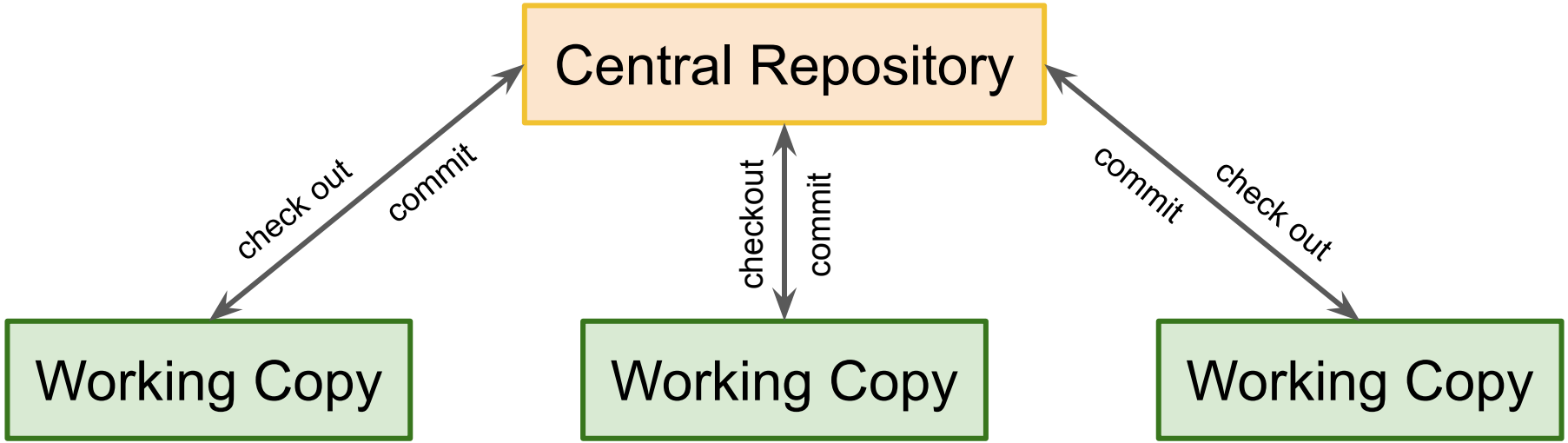
- Concurrent Versions System (CVS)
- Apache Subversion (SVN)

} Centralized Version Control

- Google Drive
- Dropbox

} Don’t use these for software

Centralized Version Control



The code lives in a central repository.

Everyone gets a copy of the current version.

Comments on Centralized Source Control (1)

- A centralized repository holds the files in both of the following models
  - This means a specific computer is required with some disk space
  - It should be backed up!

1. Read-only local workspaces and locks

- Every developer has a read-only copy of the source files
- Individual files are checked out as needed and locked in the repo in order to gain write access
- Unlocking the file commits the changes to the repo and makes the file read-only again

Comments on Centralized Source Control (1)

- A centralized repository holds the files in both of the following models
  - This means a specific computer is required with some disk space
  - It should be backed up!

2. Read / Write Local Workspaces and Merging

- Every developer has a local copy of the source files
- Everybody can read and write files in their local copy
- Conflicts between simultaneous edits handled with merging algorithms or manually when files are synced against the repo or committed to it
- CVS and Subversion behave this way

CVS: Concurrent Versions System

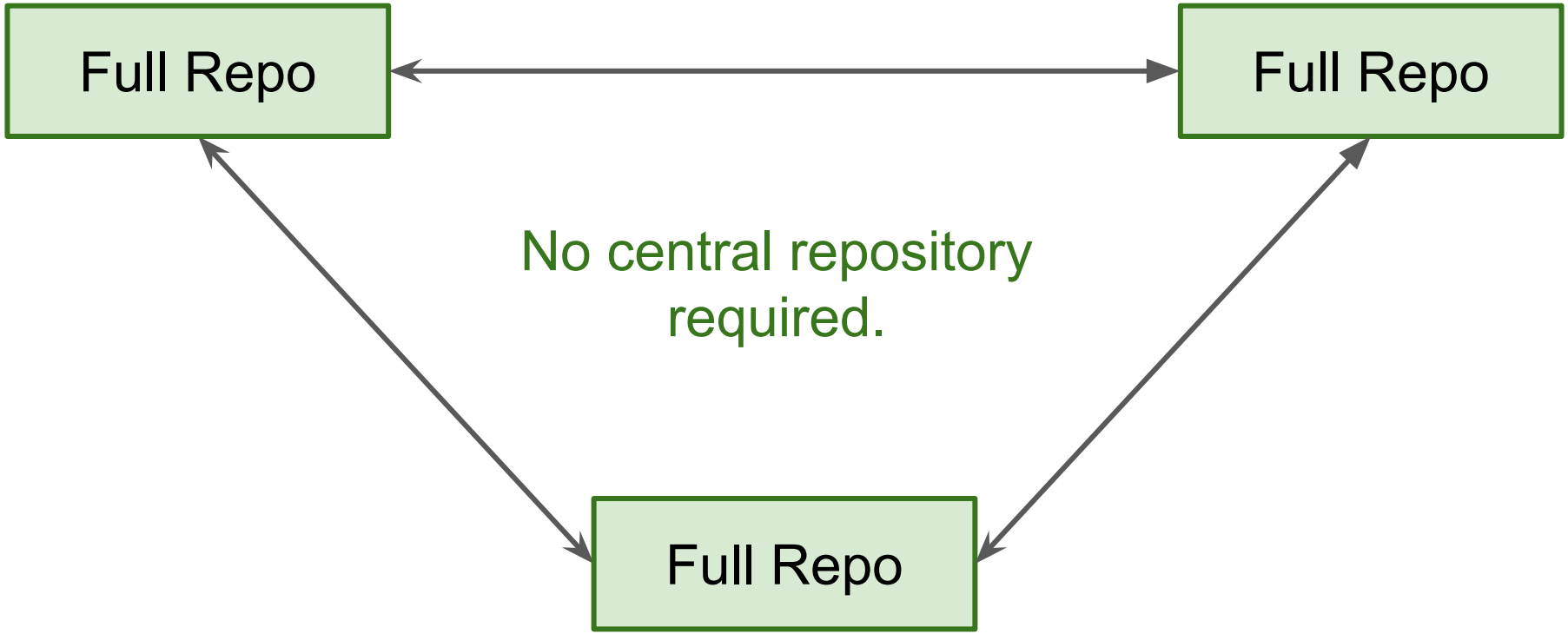
- Started with some shell scripts in 1986
- Recoded in 1989
- Evolving every since (mostly unchanging now)
  - CVS--Concurrent Versions System
- Uses read / write local workspaces and merging
- Only stores differences between versions
  - Saves space
  - Basically uses `diff(1)` and `diff3(1)`
- Works with local repositories or over the network with `rsh` / `ssh`

Subversion

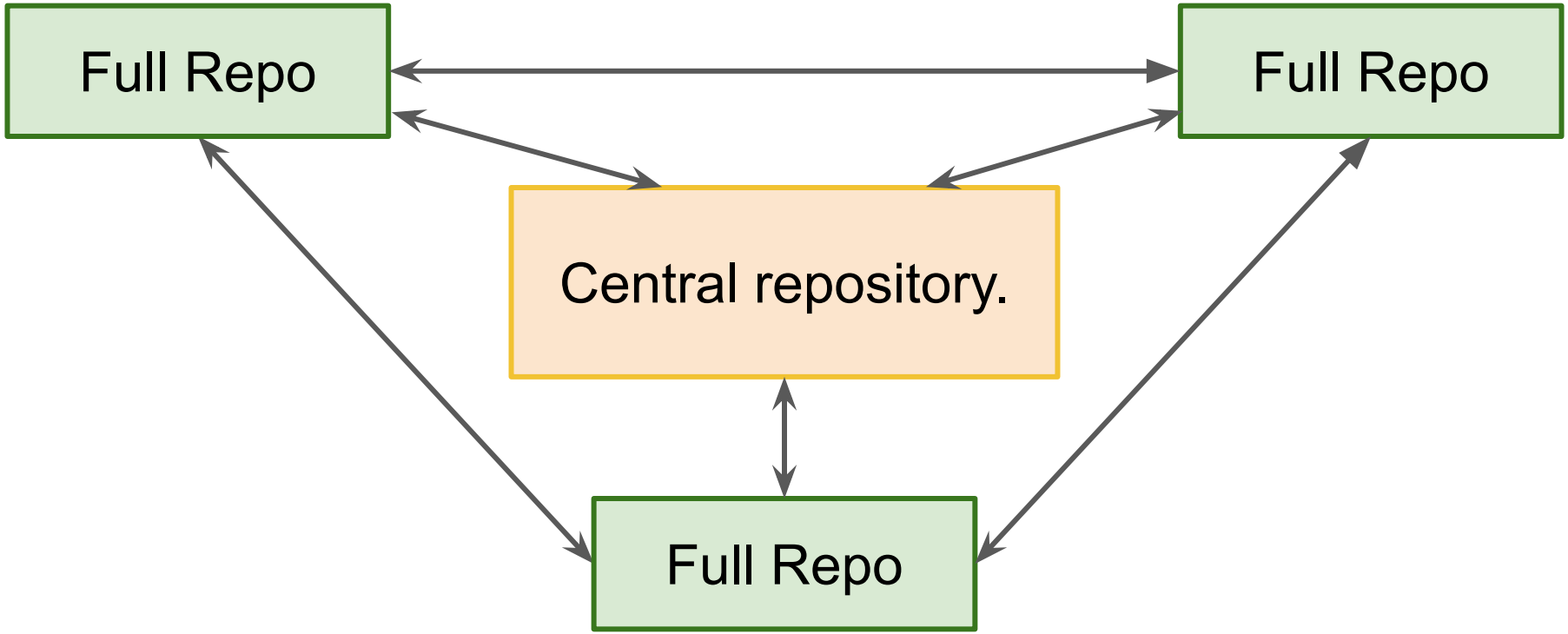
Subversion is a functional superset of CVS

- Began development in 2000 as a replacement for CVS
- Includes directory versioning (rename and moves)
- Truly atomic commits
  - i.e. interrupted commit operations do not cause repository inconsistency or corruption
- File metadata
- True client-server model
- Cross-platform, open source

Distributed Version Control



Distributed Version Control



Getting Started with Git

There are many Git tutorials

- Git for beginners: The definitive practical guide
- Learn Git- Git tutorials, workflows and commands
- Resources to learn Git
- ⋮
- Others on the course Resources page

Git was created by Linus Torvalds for work on the Linux kernel ~ 2005

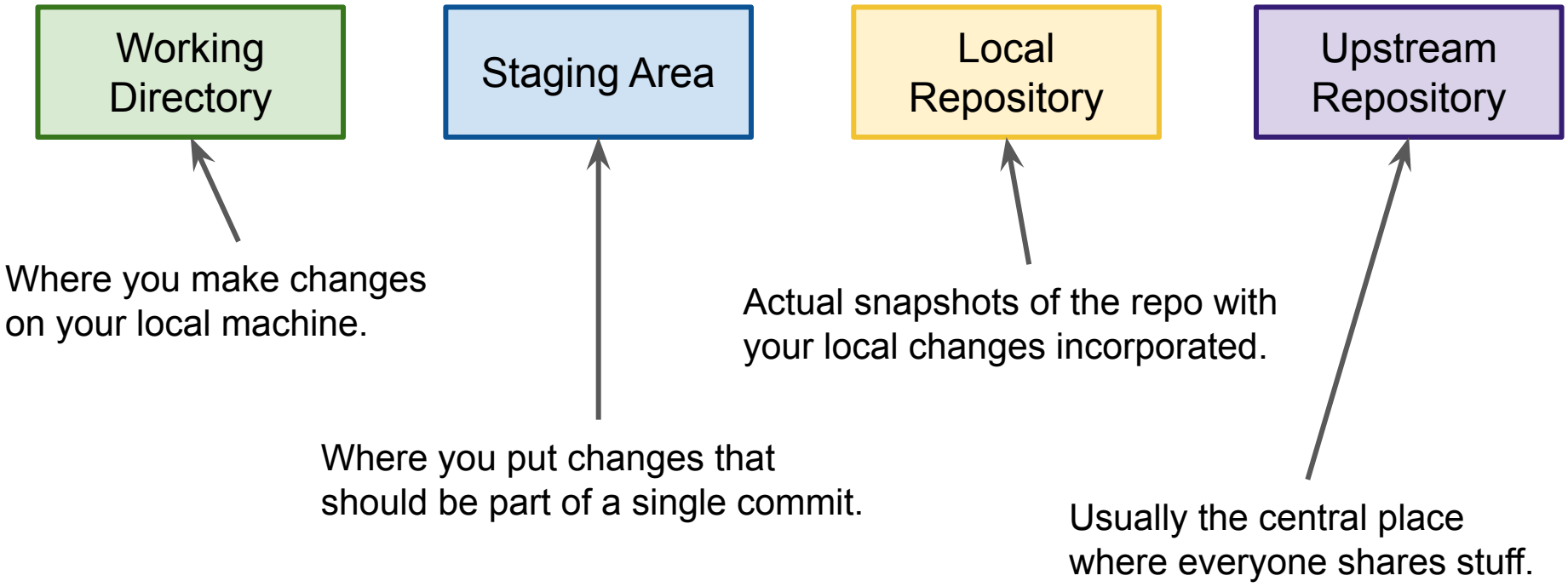
Git is...

- A **Distributed** Version Control system or
- A Directory Content Management System
- A Tree history storage system

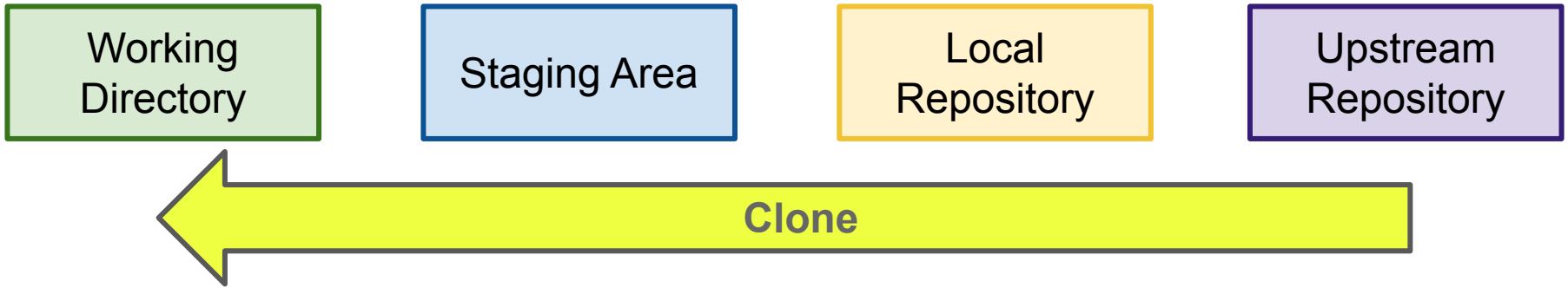
The meaning of “distributed”:

- Everyone has the complete history
- Everything is done offline
- No central authority
- Changes can be shared without a server

The Bare Essentials of Git



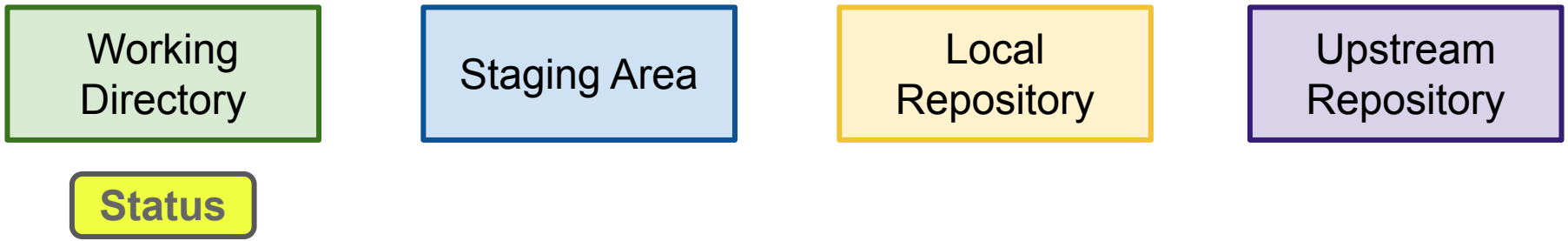
The Bare Essentials of Git



The act of getting an exact copy of the repository to your local machine.

```
git clone <repo_url>
```

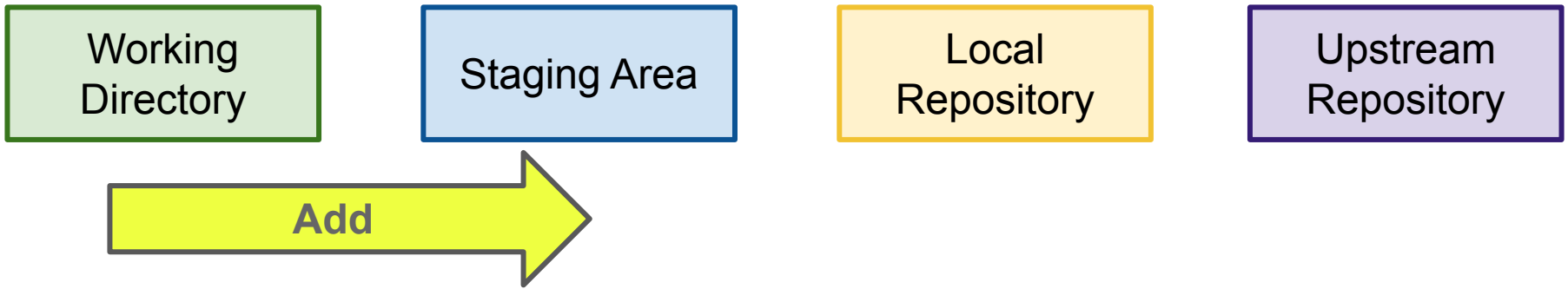
The Bare Essentials of Git



The current status of your repository. Tells you what's been changed.

```
git status
```

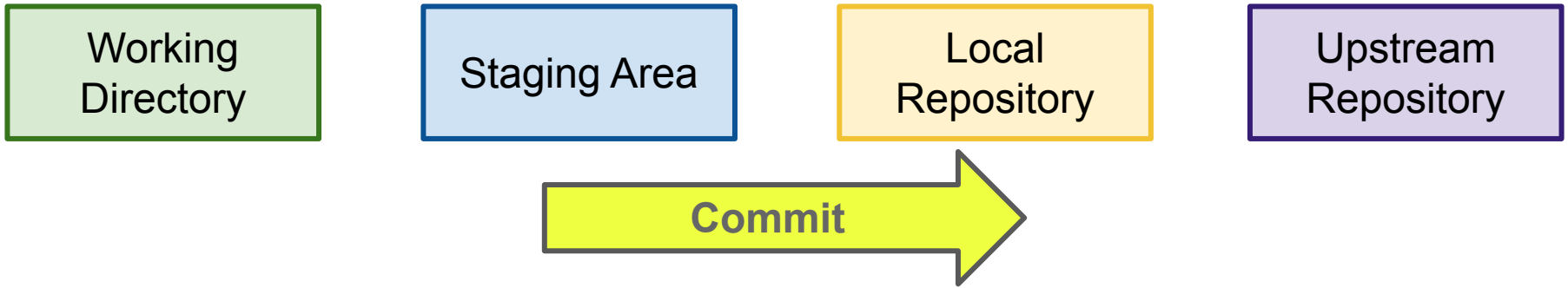
The Bare Essentials of Git



Put a new and/or changed file in the staging area.

```
git add <new_file>
```

The Bare Essentials of Git



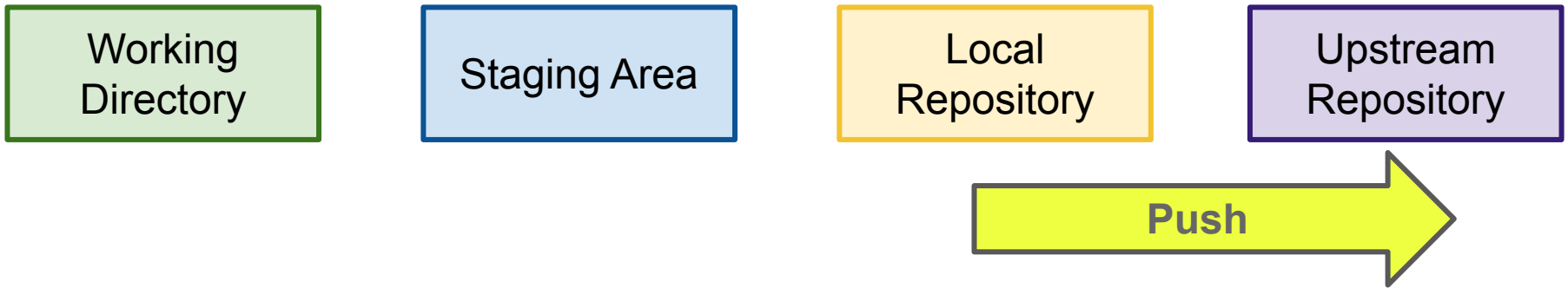
Change and update the local repository with the files in the staging area.

```
git commit

git commit -a

git commit -m “commit message”
```

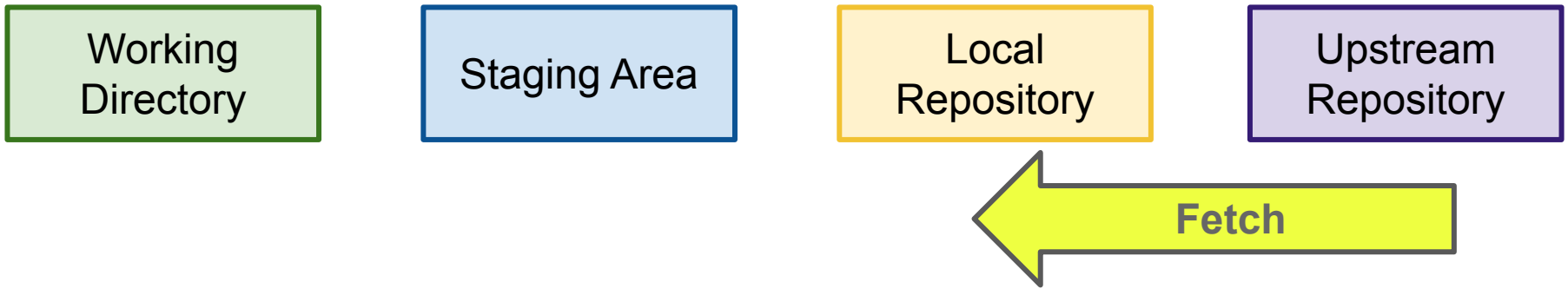
The Bare Essentials of Git



Push all commits to a remote repository.

```
git push
```

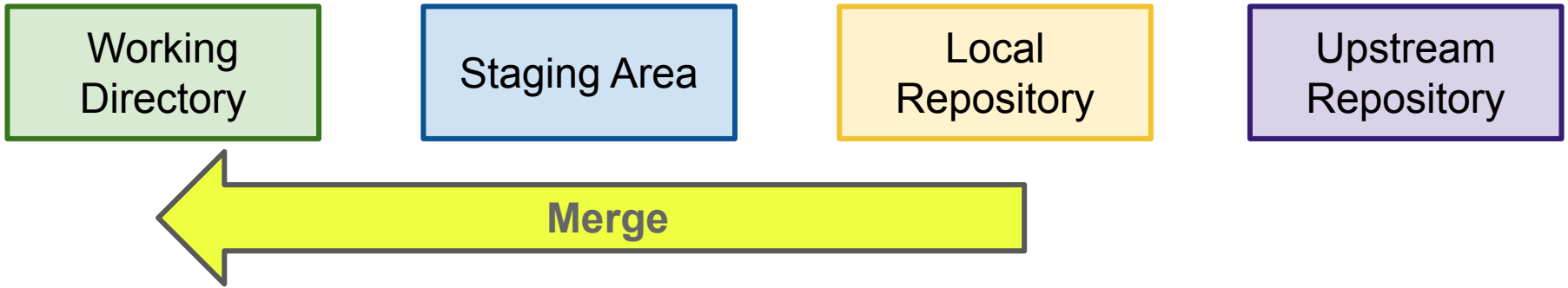
The Bare Essentials of Git



Get the changes (commits) present in the remote repo. Do not update the local repo.

```
git fetch
```

The Bare Essentials of Git



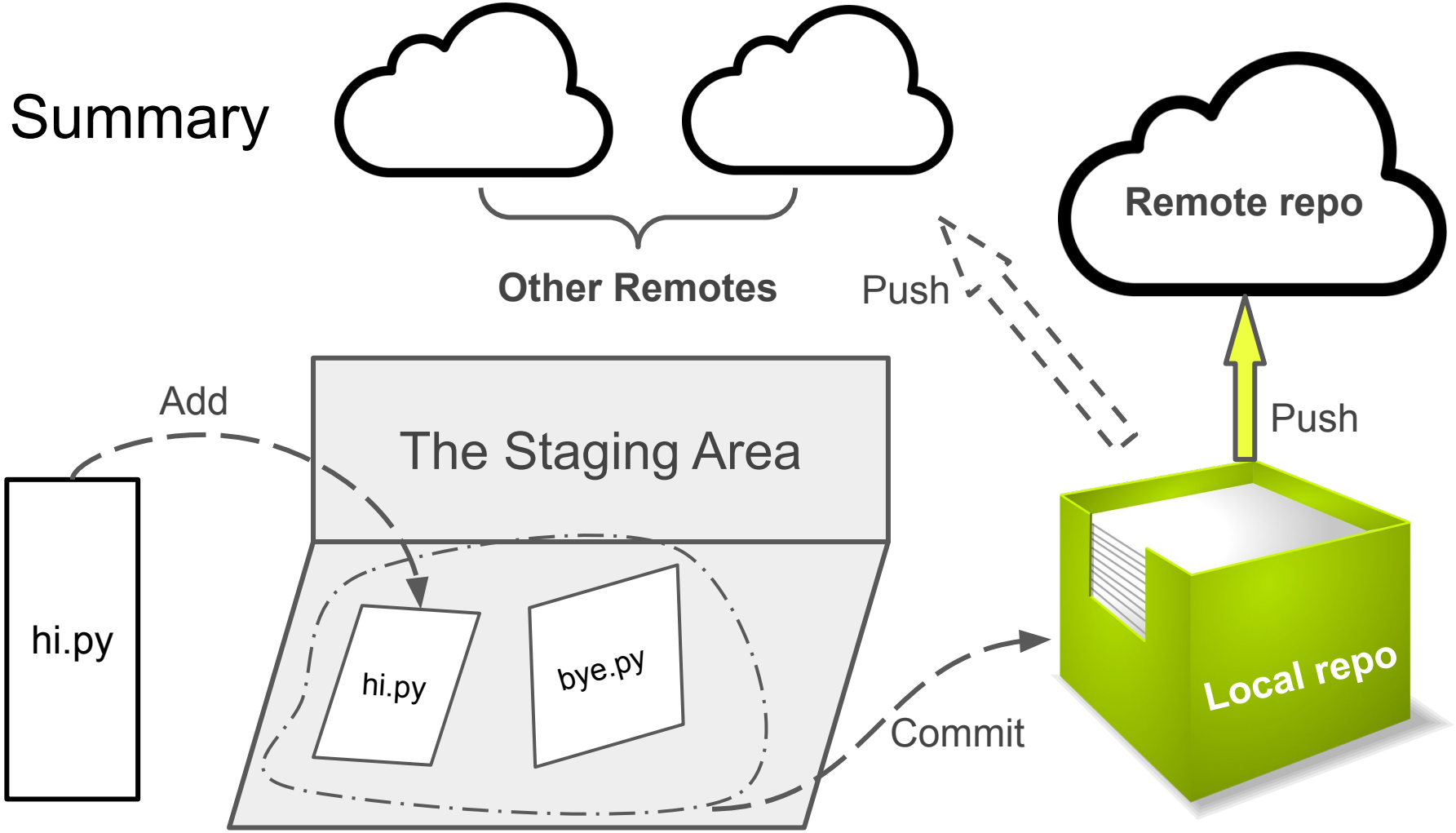
Updates the local repository with the latest changes from the upstream repo.

```
git merge
```

When to Commit

- Committing too often may leave the repo in a state where the current version doesn’t compile
- Committing too infrequently means that collaborators are waiting for your important changes, bug fixes, etc. to show up
  - Makes conflicts much more likely
- Common policies
  - Committed files must compile and link
  - Committed files must pass some minimal regression test(s)
- Come to some agreement with your collaborators about the state of the repo

Summary


Other Remotes
Remote repo
Push
Push
Add
The Staging Area
hi.py
hi.py
bye.py
Commit
Local repo

In-Class Demos