Contents

This handout is meant to give you a quick introduction on how each of the infrastructure in 6.191[6.004] works.

1 Bash

Bash is the name of a "shell" i.e. a program that allows you to interact with the Operating System (OS) and asks the OS to do tasks for you. When you log into Athena, the server will start up a Bash shell, which you can then interact with through the SSH connection. The prompt of a Bash-shell usually looks something like the following.

```
{kerberos}@ten-thousand-dollar-bill:~/some_directory_names$ _
```

Bash allows you to interact with the OS through commands that you can type into the prompt. Commands are generally formatted in the following way:

```
command [flags] arg1 arg2 arg3 ...
```

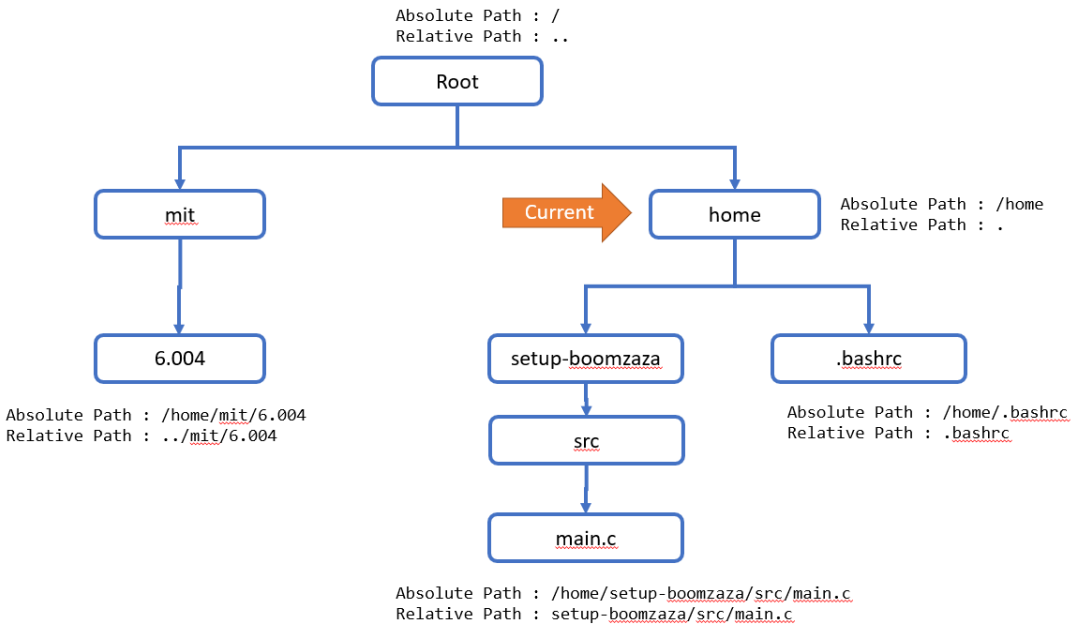
where

- `command` indicates the command to invoke.
- `arg1` and `arg2` indicate input arguments to the invoked command. Each argument is separated by spaces. If you want to pass an argument that has a space in it, you can escape the argument by using the quote sign (`'Like this'`).
- `flags` are options that can be passed in to the command. Flags are usually optional and begin with the dash sign (For example, `-al`).

1.1 File system and path

One of the most common tasks that Bash handles is to interact with a UNIX-like file system, which is generally very similar to what we have in a normal PC. All file content lives inside the top-level directory called the `root` directory, and each directory can contain more sub-directories and/or files.

In order for Bash to know what file or directory we are referring to, bash uses the notion of a **path** to locate the file. Path conventions are summarized in the image below:



Example of how to refer to each file or directory using a path

There are usually 2 kinds of path specification:

1. **Absolute path**: The absolute path shows the full path traversed from the `root` directory to the directory (or file) you are trying to access. This kind of path begins with a "/" which specifies that it is an absolute path, followed by a series of "/" separated subdirectories in the order in which they are traversed, and an optional file name. For example:

   ```
   /mit/6.004
   ```

   This path refers to a directory (or file) that can be accessed by starting at the `root` directory, then traversing to the `mit` subdirectory and looking at the directory (or file) named `6.004`.

2. **Relative path**: A relative path shows how to reach the file starting from the **current directory**. Note that while using a Bash terminal, Bash remembers which directory you are currently in and usually includes it as a part of the prompt. If you need to know what your current directory (aka `present working directory`) is, you can also type

   ```
   pwd
   ```

   to see the absolute path of your current directory.

   To specify a relative path, we start the path with directory names directly without a "/" character. For example:

   ```
   setup-boomzaza/src/main.c
   ```

   This path refers to the file (or directory) that can be accessed by starting off in the `home` directory as your current directory, then traversing to the `setup-boomzaza` directory, then traversing to the `src` directory that lives inside that, and grabbing the `main.c` file that lives inside the src directory.

Note that there are some special directory names that have special meanings when used in the path.

- `.` directory: This directory refers to its own directory. Each directory has the `.` subdirectory which will just refer to itself. For example, the path `./mit/./6.004` is equivalent to `mit/6.004`.
- `..` directory: This directory refers to its parent directory. Each directory has the `..` subdirectory which will just refer to the directory that contains it. For example, the path `./mit/6.004/..` is equivalent to `mit` because `mit` is the parent directory of the `6.004` directory.
- `~` directory: This directory refers to the **home directory** which is set according to a certain default value. For example, the path `~/setup-boomzaza` is equivalent to `setup-boomzaza`, which is in the home directory. Each user has their own `home directory`.

1.2 File system navigation

1.2.1 `cd`

`cd` commands are frequently used to change your current directory to another directory. The syntax of the command is

```
cd {path-to-other-directory}
```

For example:

- `cd ..` will change your current directory to the parent directory
- `cd setup-boomzaza` will traverse your current directory down through the setup-boomzaza sub-directory
- `cd /mit/6.004` will traverse you to the `/mit/6.004` directory
- `cd ~` will traverse you to your home directory

1.2.2 `ls`

`ls` commands are frequently used to list all files and directories in the current directory. Here are some practical uses of the `ls` command:

- `ls` will list all files and directories in the current directory
- `ls {path}` will list all files and directories in the given directory path
- `ls -a {path}` will list all files and directories in the given directory path, including ones that are supposed to be hidden (such as `.` and `..`)
- `ls -l {path}` will list all files and directories in the given directory with a long listing format (which contains information about permissions and whether the name is a file or directory)

1.2.3 Show file content

We can also use `cat` and `less` to show file content.

- `cat {path}` will show the content of the file given the path
- `less {path}` will show the content of the file specified by path with scroll support. This is useful when file content is too large to be displayed in a single screen.

1.3 File or directory manipulation

Here are some basic commands to use to create, move, copy, or remove files and directories in Bash:

- `touch {file-name}` will create an empty file in the current directory
- `mkdir {dir-name}` will create an empty directory in the current directory
- `mv {file-name1} {file-name2}` will change the name of the file (or directory) from {file-name1} to {file-name2}
- `mv {path1} {path2}` will move the file (or directory) from {path1} to {path 2}
- `cp {file-name1} {file-name2}` will make a copy of file {file-name1} in {file-name2}
- `cp -r {path1} {path2}` will recursively copy the directory and its contents
- `rm {file-name}` will remove the file. Note that there is no recycle bin in bash. `rm` command will **permanently delete** a file, and this action cannot be undone, so always triple check that you are removing the right file.
- `rm -r {dir-name}` will remove the directory and all its content. Same warning as above.

To edit a specific file, we need to use an **Editor** which is a program that will show and allow you to edit the file content. Popular options available in Athena include `nano`, `emacs`, `vim`, and `vi`. To open the editor and start editing a file, the syntax is:

```
nano {path-to-file}
```

or

```
vim {path-to-file}
```

replacing the first command with your editor of choice.

Each editor has different ways for the user to interact with it. `nano` is friendlier towards users who are new to Bash, whereas `emacs`, `vim`, and `vi` have a steeper learning curve.

1.4 Executable

Certain files in Bash are **executable**: you can run them just like a normal program. The syntax to run an executable file is

```
./{path-to-file}
```

In fact, most of the tests in the labs you will be working on are also compiled into an executable, and you can run these tests using the same syntax.

1.5 Scripts

A **script** is another kind of file that is executable. However, it relies on **another** program to read the script and execute it. Some of the most common kinds of scripts are Python scripts (where we need to ask the `python` program to read the file content and execute it), or Bash scripts (where we need to ask the Bash shell to read the content and execute it).

Bash Script content usually looks something like this:

```
!#/bin/bash
source /mit/6.004.sh
add -f git
```

The first line of a script usually tells Bash where it should look for the program to read the script (in this case, Bash). Upon executing a Bash script, Bash reads and executes the file line-by-line.

The `.bashrc` file is a Bash script that gets executed everytime you log into a Bash session. This is why when you put commands into the `.bashrc` file, you do not have to re-execute them everytime you log into Athena.

1.6 Makefile

Makefile is a simple build system that works similar to Bash scripts. The syntax to run Makefile is the following

```
make {target-to-build}
```

Upon running this command, the following things happen.

1. The command first checks for the `Makefile` file in the current directory.
2. It then tries to find a script in the `Makefile` file corresponding to the desired build target.
3. It then checks for dependencies to see whether some scripts need to be re-run again.
4. Finally, it runs all the Bash scripts needed to build the desired target executable..

This is how we allow Bash to run Minispec and test your Minispec code in the lab.

1.7 Background process

It is possible to run multiple commands concurrently in Bash. To do this, we tell Bash that we want to run a particular command in the background so that you can continue executing further Bash commands. The following commands are examples of techniques you might use in Bash to execute commands in the background.

- `{command} &` will execute the command and put it in the background.
- Pressing Ctrl-Z while running a specific command will halt the execution of that command. Subsequently, typing `bg` will allow the command to continue executing in the background. On the other hand, typing `fg` after the halt will allow the command to continue executing in the foreground.
- `jobs` will list all the active jobs (commands) that are running
- `ps` will list all the active processes (commands) that are running together with their corresponding process number.
- `kill {process number}` to terminate process {process number}.

1.8 Exploring other commands

Bash also includes some commands to help you learn more information about other Bash commands.

- `type {command}` will show you what kind of command the `command` is. Note that there are multiple kinds of command types: some of the commands are shell built-ins, some are actually stored as an executable, and some are a shell alias to another command. For example, `cd` is a shell built-in, whereas the `ls` command is stored as a binary executable somewhere.
- `help {command}` will show you details about the Bash built-in command.
- `whatis {command}` will show you a 1-line description of the command.
- `man {command}` will show you the manual page for the command.
- `alias` will show you list of all aliased commands.
- `alias {alias-name}='{command string and flags}'` can be used to create an alias command.

2 Git

Git is a simple version-control software that allows you to track changes to your code over time and allows you to easily synchronize your codebase across multiple machines. In 6.191, we use Git to manage all your lab submissions and to automatically send your code to our graders every time you update it.

2.1 Repository

A **repository** is a working directory that Git will manage and synchronize. It contains files and directories that you can open and edit using the Bash file system, but it also contains additional information that keeps track of the different versions of each of the files in the directory.

Usually in 6.191, we will be working with 2 repositories.

- **Remote Repository**: this is a repository of a lab that you will have on a github server. This will be created once you click the Create Repository link for the lab you are working on. We use this repository to grade your lab submissions.
- **Local Repository**: This is a repository of your lab that you will have on Athena. In order to obtain this repository, you will need to **clone** the repository from the remote repository. Upon running the

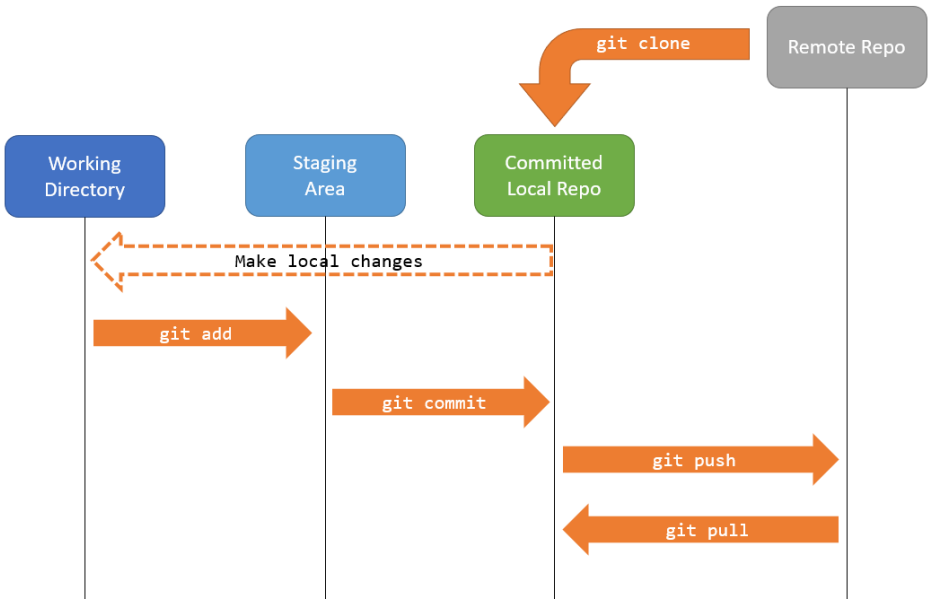
  ```
  git clone git@github.mit.edu:6191-sp24/labX-{YourMITUsername}.git
  ```

  on Athena, Git will create a local Git repository that is a replica of the cloned remote repository. Once you finished cloning a repository onto a local machine, you can then use the rest of Git commands to manage the local repository accordingly. Once you are satisfied with the contents of your local repository you can then `git push` the content to the remote repository. This workflow is described in more detail in the section below.

2.2 General workflow

One concept that is important to any Git workflow is the concept of the **commit**. **Commit** refers to a snapshot of files in the repository. Git will never change Committed snapshots unless you explicity ask Git to do so (hence the name "commit").

The following picture summarizes how Git works in general.



Summary of how Git works

2.2.1 Staging area

Whenever you make a change to any file in the Git repository, the change will not be tracked by Git yet. Git commits will only record a change made in a file if it resides in the **Staging area**. This feature allows you to work on some other source files without committing them into a Git commit. To add files to the staging area, you can use the following command:

```
git add File1 File2 File3
```

To check which files are in the staging area, you can also use the following command:

```
git status
```

2.2.2 Committing

Once you have added all the relevant files to the staging area, you can create a commit to save a snapshot of all the files in the staging area. The `git commit` command will save a version snapshot of all files in the staging area and create an identifiable point that you can come back to.

You can perform the commit by running the following command

```
git commit -m "Message here: commits only changes in File1 File2 File3"
```

An alternative to first adding all modified tracked files to the staging area and then committing them using the two separate commands `git add {List of all modified tracked files}` and `git commit`, is to run the following single command which is a shorthand for these two commands:

```
git commit -am "Message here: commits all changes on all modified files that are tracked bit git"
```

2.2.3 Pushing

In order to sync the changes you made in the local repository with the remote repository, you will have to **push** the Git repository up to the remote repository. This can be simply done by running

```
git push
```

This will push all the commit history up to the remote repository. **This does not push uncommitted changes however**, so make sure to commit your files before the push.

2.2.4 Pulling

In order to get any changes that have made it to the remote repository which you do not have in your local repository (this can happen if you have multiple local repositories on different machines), then you can **pull** from the remote repository to your local repository. This can be done by running

```
git pull
```

This will fetch and download content from the remote repository to your local repository so that the local repository is up to date on changes made elsewhere.

3 Quick Reference

Syntax of Bash command

`command -flags {arguments separated by spaces}`

- Flags indicate command options
- Flags are optional and may be omitted
- Flags must be preceded by one or two dashes, depending on the flag and command
- Single-character flags can often be combined (e.g. `ls -l -a` can be written as `ls -la`)
- To pass an argument to Bash that contains spaces, delimit the argument with single quotes 'like this'

Efficient Bash Usage

- Tab completion: Press the Tab key to auto-fill directory and file names
- Arrow keys can be used to see and run previous commands
- Ctrl-a will move your cursor to the beginning of the line, Ctrl-e will move your cursor to the end of the line
- Ctrl-c will cancel a command that has been typed but not entered (useful for quickly canceling text entry). If Ctrl-c is pressed when a command is running it will force-quit the command.

Paths

- Root directory : " / "
- User home directory : " ~"
- Current directory : " . "
- Parent directory (relative to current directory) : " .. "
- Directories are separated by the forward slash character (e.g. `/home/student/MIT/6.004/`)
- Paths can be relative (to the current directory) or absolute (entire path specified starting with the root directory)
- Use `pwd` to print the current working directory

`cd <path>`

Change working directory

- `cd` with no arguments this will change the current working directory to the user's home
- `cd -` will undo the previous cd command

`ls`

List directory contents

- `-l` : list file details
- `-a` : list all files (including hidden files)

`less <path to file>`

Display text file in isolated view with scroll support

- Press "q" to quit and return to the Bash instance

`cat <path to file>`

Print text file to the terminal

- Useful for concatenating files via the redirection operators > and »

`echo`

Print string literals or variables to the terminal

- `echo $?` will print the exit code of the previous command; useful for debugging

`file <path to file>`

Shows file's type (text, image, video, etc.)

`touch <path to file>`

Create empty file

`mv <path to source file> <path to destination file>`

Move or rename file/directory

`mkdir <path to directory>`

Make a new directory

`cp <path to source file> <path to destination file>`

Copy a file

- The `-r` (recursive) flag can be used to copy directories

`nano <path to file (optional)>`

Basic text editor and scratch-pad

- Key notation: `^` refers to [Ctrl] , `M` refers to the Meta key ([Alt] / [Option])

`rm <path to file or directory>`

Remove file.

- When you delete files (or directories) with rm they cannot be recovered from the "Trash" or "Recycling Bin", only delete files if you are sure they will not be needed in the future.
- Triple check rm commands and other dangerous commands before running.
- Avoid running commands from the Internet without knowing their function
- When using rm make sure your paths do not have spaces between forward slashes

Flags

- `-r` : remove recursively (will remove directory specified and all its contents)
- `-f` : force file removal, do not prompt before deletion

`grep -flags(optional) 'text to search' <file or directory>`

Find text in a file or across files

Flags:

- `-i` : case insensitive search
- `-r` : recursive (used to search within directories)

`find <search directory> -iname '*name to look for*'`

Find files or Directories. Unlike `grep`, find matches name and iname exactly, so you must either specify the entire file name or use the wildcard character (*) that represents any text (including no text: " ")

Flags:

- `-name` : search by file/directory name, case sensitive
- `-iname` : search by file/directory name, case insensitive
- `-maxdepth <N>` : limit search to specified recursive depth
- `-type d` : find directories

`type <command name>`

Show command type and location

`alias alias-name='<command string here>'`

Save a command shortcut

- Do not insert spaces between the alias-name, the equality sign, and the initial quote
- Must be saved in `~/.bashrc` (GNU+Linux), `~/.bashrc.mine` (Athena), or `~/.bash_profile` (Macintosh) to be persistent
- e.g. : `alias athena='ssh {kerberos}@athena.dialup.mit.edu'` will allow you to ssh into athena by running the command `athena`

`man <command name>`

Show manual page for command

`help <command name>`

Show help for Bash built-in commands

`whatis <command name>`

Show one-line explanation of command

`apropos <text>`

Show commands related to <text>

`exit`

Close Bash session / terminate SSH connection

`git`

- `git clone {remote-repo-url}` : copy the remote repository into the local machine
- `git add {file}` : add file to the stagine area
- `git commit -m "Message"` : commit all the files in the staging area
- `git commit -am "Message"` : add all the files in the repository and commit them

- `git status` : look at the current status of the staging area
- `git push`: push the repository commits up to the remote repository