Configurations, configurations, configurations

In the previous meeting, we had a somewhat detailed set of instructions for installing Ubuntu. Since the definition of our development environments is somewhat personal, this meeting will be more about exploring the specific tools that you choose to use. If you do not have a strong opinion about a particular text editor, I will make suggestions, but they are just suggestions. However, there are some tools, like `ssh`, that are essential. Please complete the tasks below. If you finish early, you may wish to start some of the optional extra tasks.

Users and Groups

In the first meeting, we created a shared account on our machines. That is fine since we are a team, and "sharing is caring". But sometimes you also need to go our own way.

- Use `less` to look at the file `/etc/group`. `/etc/group` lists all of the groups on your system. Are any of the groups familiar?
- Use `less` to look at the file `/etc/passwd`. `/etc/passwd` lists all of the users on your system (among other things). Do you see yourself?
- At the terminal, type the command `groups`. What does `groups` do?

Now please create a new user for each member of your team. You can use your first name, Williams Unix, your favorite color. . . your username doesn't matter. However, your password *must* be secure.

- Use `less` to look at the file `/etc/group`. What has changed? Why?
- Use `less` to look at the file `/etc/passwd`. What has changed?

Now open a new terminal, and switch to one of your newly created users within it.

- What is the output of `groups`?
- Type `sudo ls`. What happens?
- In what groups do you need membership to be able to use `sudo`?

Please add your new users to the appropriate groups so that they have superuser permissions. Also create a new group, called `wsp11`, and add the new users to that group as well.

Bash Customizations

As we saw in the **Shell Initialization** section of bash academy, there are several files that are used to initialize our shell when we log-in. Please edit the files `~/.bashrc` and `~/.bash_profile`. It is often common to use the `source` command within one of these files to include commands from other file(s). I like to create a `~/.aliases` file that includes all of my command aliases, and a `~/.exports` file to include all of my variables. I then use `source` to include these (and others) in my `~/.bash*` files.

At the minimum, please do the following:

- Create an alias for the `rm` command so that it forces you to confirm the deletion of files before deleting them.
- Modify the `PS1` variable to control the look of your command prompt.
- Add shortcuts for any other commands you commonly find yourself (mis)typing. For example, I have aliases for `l`, `ll`, and `la` that call `ls --color=auto`, `ls -l`, and `ls -a` respectively. Alias syntax is especially useful at Williams, since some people have been known to leave aliases as reminders to students not to leave themselves logged-in at an unattended computer.

As we saw in the **Environment Variables** section of bash academy, the `LSCOLORS` variable controls the way the listing of `ls` is displayed in your terminal. Please modify the `LSCOLORS` variable so that your file listings are satisfying rather than dull.

`ssh`

Secure shell, or `ssh`, is "a cryptographic network protocol for operating network services securely over an unsecured network". `ssh` is one of the most important tools you will use because we rarely do all of our work on one machine. In fact, I don't remember the last time that I used a computer (for work) where I did not use `ssh`. I use `ssh` to:

- log-in to a server to run experiments so that I (1) don't consume all of the resources on my local machine, (2) isolate my working environment from the inevitable system crashes caused by my experiments, (3) prevent my work from generating experimental noise, and (4) can walk away from my experiments without worrying about them.
- connect to the lab from my home
- access to on-campus resources from off-campus
- connect to a virtual machine from my host machine
- checkout `git` repositories

The basic command for `ssh` is simple:

```
$ ssh <user>@<machine>
```

A machine can be specified by its hostname (such as `panic11.cs.williams.edu`) or by its IP address (such as 137.165.8.68).
When you issue a command like:

```
$ ssh jannen@doran.cs.williams.edu
```

You are prompted for a password (when you type nothing shows up for security purposes). After successfully entering your password, you are given a prompt on the target machine.
Those are the `ssh` basics, but there is much more to know. Please explore the following questions:

- What is an `ssh` key? How do I create one?
- What is a public key vs. a private key? Where should each live?
- What `ssh` configuration files control client behavior?
- Server behavior?
- How do I create a "shortcut" entry in my client `ssh` configuration file and specify information about a remote machine? (For example, if I wanted the command `'ssh home'` to `ssh` into `busa.cs.williams.edu` with username `bill`, port number `22`, and to use the private `ssh` key `bill_id`)
- How can I configure `busa.cs.williams.edu` to accept my private key?
- How can I use graphical applications over `ssh`?
- What happens if I am running a program on some server, and my connection is severed (wifi goes down, I trip on my Ethernet cord, I accidentally close my terminal window, etc.)? Don't wory, we will learn utilities that can help in this situation.

The following commands aren't `ssh` specific, but they are often useful when using `ssh` to connect to shared remote machines.

- From the command line, what command tells me:
  - the name of the machine am I on?
  - the usernames of everyone that is logged-in?
- On a host machine, how would I broadcast a message to all logged-in users?
- On a host machine, how would I get a information about all the processes being run by a particular user?

`ssh` extras (for those that are interested)

Often there are CS resources that cannot be accessed from off-campus. You can actually use `ssh` to *tunnel* your browser's traffic to another machine so that it appears that your traffic originates from that machine. Look up how to create a Firefox or Chrome profile (so that you can make setting changes but don't always do this by default), and then on a separate profile tunnel your connection through `ssh`.

tmux and screen

Both tmux and screen are incredibly useful, and here are some comparisons. I use tmux for every ssh connection. I also use tmux locally. I almost always have at least one tmux session open on every machine.

There are many tutorials online. Please feel free to refer to them while exploring. Perform the following tasks from the command line:

- Expand your terminal window so that it is at least 160 characters wide.
- Create a new tmux session.
- Split your window vertically so that you have a "left" and a "right" side.
- Switch between both panes.
- Delete one pane, then recreate it.
- Create a new window.
- Switch between windows.
- Create a window with a specific name.
- Disconnect from your tmux session without ending it.
- Create a second tmux session.
- Switch between sessions.
- Scroll up to see past output.

Interacting with tmux directly is handy, but one of the features that I find most useful is the ability to script these commands. Please write a bash script, called tmux.sh, that performs the following tasks:

- Creates a new session with a name of your choice (your Unix?) and with a window named "main". *Do not yet attach to this session!*
- Create a new window within that session called "code"
- Create a new window within that session called "compile"
- Create a new window within that session called "debug"
- Select the window 0, still without attaching.

tmux, like other programs, has its own configuration files. The tmux config file is located at ~/.tmux.conf. For some versions of tmux, when you split a pane it creates the new pane with the current working directory of the original tmux session. This is inconvenient, and the behavior can be changed in your tmux config:

```
bind '"' split-window -c "#{pane_current_path}"
bind % split-window -h -c "#{pane_current_path}"
bind c new-window -c "#{pane_current_path}"
```

There are other configuration changes I have found useful. At the minimum, please edit your tmux configuration file to:

- Limit the size of the log history.
- Turn on "aggressive-resize" mode.

You may also want to investigate tmux plugins using TPM. I use a package that lets me write my tmux history to a log-file. This is really useful If I want to keep track of output.

tmux Extras (for those who are interested)

I have found tmux useful for remotely pair programming. Please create a tmux session as one user that a second *different* user can attach. Have both of the users you created attach to this session from remote lab machines. This may require creating a Unix socket and changing file permissions.

Text Editors

One downside of using advanced text editors is that they are not terminal-based. You can often use X11-forwarding to display graphical applications, but servers *do not have X11 to forward*. For this reason, it is important to be familiar with one or more "simple" text editors. Since most of the department documentation is in `emacs`, and I am an `emacs` user myself, this section will suggest `emacs`. Please feel free to use `vim` or `nano` if that is your thing.

If you are not already familiar with `emacs`, that is the first step. Get familiar by reading the documentation above, or by reading some of the resources in the ebooks directory. There is an `emacs` reference book (as well as a reference book for `vim`).

The standard `emacs` config file is located in `~/.emacs`. Editing `emacs` configuration files uses the programming language LISP, but you can pattern-match. Please edit your `emacs` config to add the following features:

- Show the time and date
- Highlight matching parentheses
- Highlight the region when a mark is set (when you hit `Ctrl-space`)
- Show the row and column numbers of your cursor
- Disable the startup message (maybe called the splash screen?)
- Disable the startup message about the scratch buffer
- Add a new key-binding of your choosing. One idea might be to bind a key to jump to a specific line.

`emacs` Extras (for those who are interested)

- `emacs` also supports packages. Edit your config file to `(require 'package)`. Then find and add a package that you think is interesting.
- As an `R` user, the absolute best thing is `ESS`, or `emacs` speaks statistics. It lets you edit R code within `emacs` and also execute particular lines, regions, or files. You can install the ess package in Ubuntu using `apt-get`. Set up ess and use `M-x R` to start a session within `emacs`.
- You can run `bash` within `emacs` in a variety of modes. Try executing `ls` within `emacs`, and go from there.
- You can also run a python interpreter from within `emacs`. Try to split your window so that you are editing python code in one window and running the interpreter in another.
- One thing I have always wanted to do was use `emacs` for my email client. . . .
- `emacs` supports many interactive games. Explore one.

`git`

Now that you have edited so many configuration files, it is hard to keep track. In addition to using our wiki to document the process, we want to save the actual configuration files so we can bring them to whatever machines we work. We will use `git` to create a repository of scripts and configs. If you are not familiar with how to *use* `git`, please become familiar. This is a really nifty interactive tutorial.

Many of you are perhaps familiar with `github.com`. GitHub is great. However, `git` exists independently of GitHub, and it is important to be able to create and control your own repositories because they are private, and you and/or future employers might really care about that. Plus, like everything else, `git` has its own customizations that make it great.
The first thing you must do is install git if you haven't already.

```
$ sudo apt install git
```

Next, you are going to want to make global configuration changes for *your user*. Each user has a global configuration in `~/.gitconfig`. Here you will specify things—like your default text editor, email address, and name—that are common to all of your `git` repositories. You can always override them in any individual repository. Follow these instructions to set up your global config files for each user.

In principle, `git` is a truly distributed version control system. However, it is common to create one central repository and have all users commit their changes. To do this, one user should make a repo directory. (perhaps `/home/repositories`). Remember, we added all team-members to the same group at the beginning—this is why. Use `chmod` and `chown` to make sure that the entire group has permissions to that folder.

Now create a shared *bare* repository for your configuration files.

```
$ git init --bare --shared=group ...
```

If you worked on shared config files, add them here. If you disagreed on certain settings, create a private `branch` for each user. Organize your repository in a sensible way.

Inside each repository, there is hidden git directory called `.git`. Inside the `.git` directory is a file named `config`. Whenever you create a branch or add a remote repository using a `git` command, that information is encoded in this config file with simple text. Try some of these actions and see how the config file changes.

The `.gitignore` file is a *very* useful file. In it, you can place patterns that are excluded when you type `git status`. Each directory's `.gitignore` overrides the file in its parent directory; in other words, you can specify files to exclude, and then include them in a child directory. I often find myself excluding the backup files generated by my text editor (`*~` in `emacs`). I also often exclude files that are automatically generated. For example, if I am writing Java code, I excluded the `.class` files; in C, the `.o` files; and in Python, `.pyc` and the `venv/` directory. Create a `.gitignore` file in a parent directory, add entries, and see what happens. Then create a child directory and override one or more of the rules.

`git` Extras (for those who are interested)

- Read this if you are curious how `git` actually works, instead of just how to use it (this is super interesting!).
- So far you have customized the `git` user configurations. You can also customize a repository with *hooks*. Create a git hook so that each time a user pushes a change to the repository, it sends you an email.