Basic - Intro
Welcome, Setup, and Some Light Reading

EECS 201 Winter 2025

Submission Instructions

This homework will be submitted via the EECS 201 Autograder. This will become evident as you work through the assignment.

Preface

Homework in this class can sometimes be a little underspecified. You are expected to Google, to try things, and to fail from time to time. Making mistakes is highly encouraged, it's how you learn. We have many office hours if you find yourself getting stuck, but we will always start with the questions, "What have you tried so far?" and "Why do you think that didn't work?"

Whenever you are prompted to run a command in assignments for this class, **type out the command** rather than copy and paste them. The act of actually typing them out is helpful for most people in picking up the syntax and being able to put together your own statements as opposed to blindly copying and pasting magical incantations.

A note on notation

Sometimes I'll use code formatting e.g. `this be some code`: this is to highlight some things that might be entered in a script or code.

I'll also use a shell prompt variant e.g. `$ execute arg1 arg2`, which is to inidicate a full command that you should enter in at the shell and run. Don't include the `$`: it's there to indicate the shell's prompt.

Lastly, I may notate some parameters like so: `<required thing>` and `[optional thing]`, and you'd replace them with necessary text minus the brackets. For example, I may say `https://gitlab.umich.edu/<your uniqname>` and for me I'd put `https://gitlab.umich.edu/brng`.
As another example I could say `$ somecommand <input file> [output file]` where you don't necessarily need to provide an output file path (with the command perhaps using a default path), and you could put `$ somecommand somefile` or `$ somecommand somefile someoutputfile`.

1 Getting access to a *nix command line

While I've set up a server for the course to serve as a reference environment, it's still advantageous to have some sort of *nix environment local on your machine with which you can use.
The following steps will depend on what operating system you are on and what sort of solution you want to take.

macOS

Congratulations! macOS is a Unix-derivative, so you're already set!

Windows 10 and 11

Fortunately in this day and age, Windows is no longer a sore spot in a landscaped filled with Unix-likes now that it has the ability to *run Linux inside of itself* via **Windows Subsystem for Linux (WSL)**.
You may hear WSL and WSL2 being tossed around: the first iteration, WSL (I'll refer to it as WSL1 from now on), is more of "pretend" Linux that's oriented around being able to run Linux executables via a compatibility layer.
WSL2 is a much more complete solution that actually runs a fully functioning Linux system under Windows. In a lot of cases it's much faster and unsurprisingly has few-to-no compatibility issues with Linux. Fortunately, WSL2 is now installed by default whenever you install WSL! If you happen to be on WSL1, I suggest upgrading to WSL2 unless there's some reason you actually need to be on WSL1. There's instructions on how to do so in the Microsoft docs. There you can also find additional information such as switching between WSL1 and WSL2, reasons why you'd actually prefer WSL1, and more.

I also encourage you to install Windows Terminal. It's a much better alternative to whatever the Ubuntu WSL app packages in, and Windows Terminal integrates with WSL very well.

Be aware that you'll have two filesystems going: one that's owned by WSL and the rest of the Windows filesystem (C: and friends). From the WSL side of things, you can access Windows files under `/mnt/c` and the like. To access WSL files in Windows Explorer, in WSL invoke `explorer.exe` for that directory e.g. `$ explorer.exe .` to open an Explorer window for the current directory, or you can use `\\wsl$` for the path.

Windows $<$ 10

If you are using an older version of Windows, your options are sadly a bit more limited and less well integrated.

Cygwin is a project that provides a runtime environment that emulates a Unix-like environment and can handle a lot of cases in this class. This is one of the things that I used before WSL was a (well functioning) thing.

You can also install a **virtual machine**, which will allow you to set up a virtual computer that you can install Ubuntu on. See the end of this document for the old VM instructions that I've kept for posterity. I also used this in the dark times before WSL was in a more usable state.

Other

Chances are this means you're running Linux, FreeBSD, or GNU Hurd, in which case you're already running a Unix-like. Huzzah!

For the adventurous: dual booting

To do this you'll have to set aside space on your hard drive to install another operating system, allowing you to boot from your original or another operating system. If you want to go down this route, I'll let you figure it out: it's a journey where you can learn quite a bit!

Virtual machines

Arguably, installing a virtual machine will more or less involve the same steps as installing an operating system for dual booting: dual booting just makes things more *physical*. Instead of installing the OS on a physical computer, you're installing it on a *virtual* computer that emulates a full computer system and runs on top of your existing OS. You can check out the end of this document for the old VM installation steps used in this class, which I've saved for posterity.

If all else fails: SSH clients

If you're unable to get access to a Unix-environment locally, you can use an SSH client like PuTTY or MobaXTerm and just use the course server. This comes at a cost of needing to be connected to the internet and reduce the opportunity to integrate Unix tools into your local workflows.

2 Playing around

Now that you have a Unix-like environment to work in, I encourage you to play around a bit in the terminal. (Feel free to skip this if you're confident in your abilities).

1. Try navigating around directories with `cd` e.g. `$ cd Documents`, `$ cd ..`. Note that if you run `cd` with no argument, most shells will take you to your home directory.
2. Try listing the non-hidden contents (files/directories starting with a `.` are hidden by default) of a directory with `ls` e.g. `$ ls`, `$ ls Documents`, `$ ls ..`
3. Try listing *all* contents of a directory with `ls` e.g. `$ ls -a`
4. Try creating a directory with `mkdir` e.g. `$ mkdir testdir`
5. Try creating an empty file using `touch` e.g. `$ touch somefile`
6. Try deleting the file with `rm` e.g. `$ rm somefile`
7. Try deleting an empty directory with `rmdir` e.g. `$ rmdir testdir`

3 UMVPN

As a result of the cyberattack and ensuing network outage just prior to the Fall 2023 semester, some network policies have changed. Most significant to this class is blocking SSH connections from outside the university network. This has affected SSH access to the EECS GitLab server (used in this class to submit coding assignments) and access to the course server (used in this class for you to run the autograder). When you are connected to a Wi-Fi network on campus, everything is fine, **but if you live off campus or decide to work on stuff off campus, you will be unable use SSH with these resources**. To get around this, you can install a VPN (virtual private network) to let you get connected to the campus network and thus get connected with the EECS GitLab and course server. You can find instructions to do this here.

For the Linux users out there who don't want to just install random software untracked by a package manager, you can utilize NetworkManager and OpenConnect (this is my setup) if it is provided by your package manager, and create a Cisco AnyConnect/OpenConnect VPN connection. UMVPN utilizes `umvpn.umnet.umich.edu` as the gateway, and you can supply something like `AnyConnect Linux_64 4.10.07061` as the user agent (source: OpenConnect on the Arch Linux wiki. If you want another VPN setup on Linux you're on your own, though I expect that if you're running Linux and this particular about how to get your VPN set up, you've already got the skills to figure that out ;)

For macOS users, if you prefer to use the built-in VPN software UM ITS also provides instructions to get it set up.

4 Course server

I've set up a course server for this class to provide an Ubuntu 22.04 reference environment. This environment is the exact environment in which your coding/scripting assignments are graded. In this environment I'll also be providing scripts that will allow you to test out your submission to see it if it will work and allow you to ask the EECS 201 autograder to grade your assignment (with unlimited reruns, though with a minimum time between runs). This server is only available for students currently enrolled in the class or who have otherwise made arrangements with the course staff. However, this server requires you to set up an SSH key to access it.

4.1 SSH and setting up an SSH key

SSH, or Secure Shell, is a protocol allowing for secure connections to another system. This allows for you to log into another computer in a secure fashion to use its shell or to use some other service.

In a lot of systems, you won't be allowed to simply type in your username and password. In lieu of this, many systems take advantage of SSH keys. SSH keys are used to authenticate you when you access a particular service. They're created as a pair, a private and a public key, for each system that you're connecting from (e.g. I have multiple computers, so I've generated keypairs for each of them). The private key is one that you keep on your local machine and hide from others: it's what's verifying that you are you. You share the public key with services that you want to access, for instance GitHub or my server, and they keep that public key on hand for your account. When you try to access the service, they send a message encrypted with your public key to your computer, your computer uses the private key to decrypt the message, and if the decrypted message matches up you're authenticated. *If someone else gets a hold of your private key, they can use it to authenticate themselves with whatever you've given your public key*. You can share one public key with multiple services: for instance, I shared my laptop's public key with GitHub, the EECS GitLab, my lab's server, and more so I can access each of them from my laptop.

Setting up a keypair is actually really easy:

1. The utility to generate a keypair is `ssh-keygen`. There's additional options that you'll need

   (a) `-t <key type>`: this specifies the type of key. **The recommended type nowadays is ed25519, which is smaller, faster, and more secure than the old RSA keys.** If your system does not support creating these keys, an RSA key will work fine as well.

   e.g. `-t ed25519`

   (b) Optional: `-C <comment>`: this specifies a **c**omment. You might've seen a lot of examples on the internet putting an email-address-like thing. More accurately, it's more of a <username>@<hostname> thing, where a hostname is like the name for your computer. The comment is just that: put whatever you wish here to help identify this key. For me, it helps me keep track of which public key belonging to which of my several computers.

2. e.g. `$ ssh-keygen -t ed25519 -C brandon@mydesktop` (be sure to fill in whatever information that pertains to you in the comment!)

3. There will be multiple prompts asking you to put in information: just leave them be and use the defaults by hitting enter/return *unless you know what you're doing*. **Leave the directory you're going to save the key in the default unless you know exactly what the reprecussions of changing it are!** If you want to have a passphrase for accessing your SSH key feel free to do so, just be aware it can cut down on some automation opportunities. If you just want the simplest set up to get SSH keys going, just hit enter/return twice without filling any info out.

4. If you peek in your home directory under `.ssh` you should see your newly created keypair! `$ cd ~/.ssh` will take you to your `.ssh` directory, and `$ cat id_ed25519.pub` will print out your pubkey. It will look something like:

   `ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIPLe4kBJcjpwnf3yq4/8aKop3MH brandon@celeritas`.

   The `ssh-ed25519` part is important. Whenever you need to supply a pubkey, be sure that it's included: for safety's sake copy and paste the entire printout of the pubkey!

4.2 Getting access to the server

Click on this link to get to the course server account form. Put in your uniqname and submit it, and the course server will email you further instructions to create/manage your user account on the course server.
Once you complete setting up your course server user account you can use `ssh` to connect to and log into the course server for a shell session on it: for me I'd run
`$ ssh brng@peritia.eecs.umich.edu`.
You'll want to run something along the lines of `<uniqname>@peritia.eecs.umich.edu`, where `<uniqname>` is your uniqname and `peritia.eecs.umich.edu` is the domain name of the server.

4.3 The EECS 201 autograder

This course uses an automated grading system for many assignments, called the "EECS 201 autograder" for brevity. This is not to be confused with the Autograder that other courses in the EECS department uses, and is one specific to this course. The EECS 201 autograder is run from the command line on the course server `peritia.eecs.umich.edu`. Log into the course server and run the following command:

```
$ eecs201-test
```

This is a program on the course server to run the EECS 201 autograder for a given assignment. With no arguments it shows what assignments are available to be graded.

Run `$ eecs201-test basic-intro` to run the EECS 201 autograder for this assignment.

This will run the EECS 201 autograder and give you credit, posting it to Canvas. Future assignments using the EECS 201 autograder will be run in a similar fashion, albeit with the appropriate assignment name for as an argument. With this, you will be tooled up to get your assignments graded in this class!

5 Reading

Here's an interesting blog post about the computing landscape and how we've grown up in it. Keep in mind that it was written over 20 years ago!

Biculturalism by Joel Spolsky
http://www.joelonsoftware.com/articles/Biculturalism.html

Posterity (100% optional): Set up an Ubuntu virtual machine feat. Virtualbox

(This is from a previous iteration of the class)
Not every *nix system is built the same way, so not *everything* will be 100% compatible. Ubuntu 22.04 will serve as the golden standard of behavior for the assignments in this class. While most things should work the same way on macOS and Ubuntu, sometimes when POSIX (the standard for *nix systems) does not specify a particular behavior for particular tools, the behavior of the tools may differ.

One of the reasons is for this is that macOS uses the BSD versions of classic *nix tools while Linux usually uses the GNU versions. BSD and GNU may have their own extended behavior in their implementations of classic *nix tools.

What a virtual machine does is provide you an emulation of a computer system. In the context for this class, the virtual machine is an emulation of a traditional desktop/laptop computer. This allows us to install "guest" operating systems on them, mess with them, and wreck them without affecting the original "host" operating system. By having a virtual machine in this class, you can freely explore the *nix environment with no danger to your "host" system.

1. Download a copy of the **Desktop** version of **Ubuntu 22.04**. This will come in the form of an *.iso* disk image file. Traditionally, you'd install an operating system with a CD or DVD; these files represent the data on those.

2. Download and install the latest version of VirtualBox. If you prefer another virtualization software (e.g. VMWare Player) feel free to use it but don't expect official support on it from the instructional staff. *If you are on Windows and using WSL2 or some other Hyper-V application (e.g. Docker), the latest version of VirtualBox should work alongside it now (just tested it personally). However, if you do run into issues, first, try going into your VM's System settings > Acceleration > Paravirtualization Interface, and set it to Hyper-V. If it still doesn't work, try taking a look into disabling Hyper-V when using VirtualBox. Here's a workaround to turn off/on Hyper-V on StackOverflow, using Command Prompt in Administrator mode. Make sure to restart your computer for this to take effect.*

3. Set up a new virtual machine.

   (a) Note that the drop-down menus for operating systems don't install the operating system; they just set up the virtual machine with some default settings and give it a cool icon. Select the "Ubuntu (64-bit)" drop-down item.

   (b) For the most part the defaults are fine. Note that the Ubuntu download page also provides some nice recommended system specs.

   (c) Ubuntu 22.04 requires at the *minimum* 2 GiB (2048 MiB) of RAM, so set the memory size to at least 2 GiB. If you can spare the RAM on your computer, I'd go with 4 GiB.

   (d) I recommend setting at least 2 CPUs to make things a bit more zippy.

   (e) Note that RAM and CPU count can be configured after setup.

   (f) The default hard drive size of 10 GiB can work, but is a tad small, leaving only about 3 GiB left on a Minimal Installation. Try maybe 20 GiB or even 50 GiB. By default, disk images are *dynamically allocated*, which means the disk image will grow on demand when more space is needed until it hits the limit you have set.

   (g) After setting up go to your VM's Display settings and max out "Video Memory". This will allow for fancier graphics like larger display resolutions to work.

4. Install Ubuntu on your new virtual machine.

   (a) Remember what I said about *.iso* files? Our VM has a virtual CD/DVD drive. Go to your VM's Storage settings and click on the Empty CD under "Controller: IDE". On the right hand side, under "Attributes" click on the CD. This will allow you to navigate and find that Ubuntu *.iso* file.

   (b) Start up your VM!

   (c) I recommend doing a "Minimal Installation" since you probably won't be consuming media on this virtual machine. Feel free to do a normal one if you do want to see what the full "Desktop Linux" experience is like.

   (d) I recommend "Downloading updates while installing".

5. Once Ubuntu is running, install the Guest Additions: check VirtualBox's Devices menu → Insert Guest Additions CD Image, then either let the disk auto-run or run the disk manually when it's recognized by Ubuntu. You might get a message about gcc, make and perl not being installed. This means you need to install those utilities. Bring up a Terminal window (right click on the desktop, look for the application, explore! There's also a keyboard

shortcut ;)) and run `$ sudo apt install gcc make perl`. This will invoke the package manager, APT, to retrieve those software packages. You can then go open the disk and click the "Run Software" button to run the installer again.

Guest additions are a utility that allow for the guest to communicate with Virtualbox and the host system. This leads to neat features like shared folders/directories, shared cliipboards, and more!

Once you install them, you should see something magical happen :) If nothing happens, try resizing the window and try maximizing the window, or try full-screening. If something still doesn't happen, turn off the machine and starting it again. If *still* you don't notice anything, turn off the machine and change the "Graphics Controller" option to something else in your VM's Display settings and try again

6. Play around with your new machine! Try installing stuff with APT, like Git: `$ sudo apt install git`. Try writing and running a Hello World program. What about other tools you've used before? If you have any, can you get an old course project running?

7. It may be helpful to setup some shared folders so you can share data between the host (your computer) and guest (your VM) operating systems. If you go to your VM's settings, there should be a "Shared Folders" section. Here you can pick out what folder on your host system to share. For a simple setup, you can select your folder, set "Auto-mount" on, and set "Make Permanent" (don't worry, you can turn this off or remove the folder sharing). In your VM, if you left the mount point blank, you can find your shared folder in `/media/sf_<your folder name>`. Try `cd`'ing or `ls`'ing it. You'll find that you don't have permissions! With `$ ls -l` we can check more info about that directory. The problem is that the user owner of the directory is `root` and the group owner is `vboxsf` ("VirtualBox Shared Folders"). This is a simple enough fix: you'll have to add your user to the `vboxsf` user group. I'll leave this as an exercise for you to figure out :) The changes are not immediate. You'll have to restart your VM. Once you're back in, try playing around with your shared folders :) If you're interested, you can find more info about VirtualBox shared folders here.