Startup Engineering
Balaji S. Srinivasan
Stanford University

Class: class.coursera.org/startup-001
Signup: coursera.org/course/startup
Forums: bit.ly/startup-eng-forum

Linux and Server-Side Javascript (SSJS)

Overview

You are likely a user of the Windows or OS X operating systems. As you may know, the operating system (OS) interfaces with a computer's hardware (like the CPU, hard drive, and network card) and exposes it via API to running programs. The OS layer thus allows an engineer to write bytes to a hard drive or send packets over a network card without worrying about the underlying details.

Most end users interact with their operating system via a graphical user interface (GUI) to watch movies, surf the web, send email, and the like. However, when your goal is to *create* new programs rather than simply *run* them, you will want to use an OS which is primarily controlled by the command-line interface (CLI), usually a Unix-based OS. Roughly speaking, Windows is mostly used for its GUI features, Linux mostly for its command line features, and OS X (built on BSD) for both. The convention in Silicon Valley today is to use OS X machines for local development and daily use, with Linux machines for production deployments and Windows machines only for testing IE or interfacing with certain kinds of hardware (e.g. Kinects). By combining OS X and Linux in this fashion one can get the best of both worlds: a modern desktop interface and a command line locally via OSX, with a license-free and agile server OS via Linux.

Now, before we go on it's worth drawing a distinction between Unix, Linux, BSD, and OS X. Unix is the granddaddy of them all, going back to AT&T. BSD is a lineal descendant of the original Unix, while Linux began as an API-compatible clone that had no shared code with the original Unix. And Mac OS X is a proprietary OS built on top of BSD which adds many new features. Today there are an incredible number of Linux and BSD variants, and you can trace through the evolution of the operating systems here. For the purposes of this class, we'll be using the Linux distribution called Ubuntu for development.

Key Features of Linux

In general, we're not going to get into low-level Linux details here, though you will need to if your app becomes wildly successful or uses significant hardware resources. The main things you need to know about Linux are:

1. *Command Line Interface*: Linux variants are built to be controlled from the command line. Every single thing that you need to accomplish can be done by typing into a command line interface (CLI) rather than clicking on buttons in a graphical user interface (GUI). In particular, it is possible to control a Linux box simply by typing in commands over a low-bandwidth text connection rather than launching a full-blown windowing system with image backgrounds and startup music.

2. *No Licensing Fees*: Linux does not require paying licensing fees to install. In theory, you can run a fleet of 1000 machines without paying any vendor fees, unlike Windows.

In practice you will need to pay for maintenance in some fashion, either via support agreements, in-house sysadmins, or by renting the computers from a cloud provider.

3. *Open Source*: Linux has a massive base of open-source software which you can download for free, mostly geared towards the needs of engineers. For example, you will be able to download free tools to edit code, convert images, and extract individual frames from movies. However, programs for doing things like *viewing* movies will not be as easy to use as the Microsoft or Apple equivalents.

4. *Server-side Loophole*: Linux is licensed under the GPL, one of the major open source licenses. What this means is that if you create and distribute a modified version of Linux, you need to make the source code of your modifications available for free. The GPL has been called a viral license for this reason: if you include GPL software in your code, and then offer that software for download, then you must make the full source code available (or risk a lawsuit from the EFF). There are workarounds for this viral property, of which the most important is called the server-side or application-service provider (ASP) loophole: as long as your software is running behind a web server, like Google, the GPL terms do not consider you a *distributor* of the code. That is, Google is only offering you a search box to use; it's not actually providing the full source code for its search engine for download, and is thus not considered a distributor. A new license called the AGPL was written to close this loophole, and there are some major projects like MongoDB that use it.

5. *Linux and Startups*: The ASP loophole is what allows modern internet companies to make a profit while still using Linux and free software. That is, they can make extensive use of open source software on their servers, without being forced to provide that software to people using their application through the browser. In practice, many internet companies nevertheless give back to the community by open sourcing significant programs or large portions of their code base. However, it should be clear that the server-side exception is a structural disadvantage for traditional "shrink-wrapped" software companies like Microsoft, for whom the GPL was originally designed to target. By distributing software on a CD or DVD, you lose the ability to use open source code, unless you do something like making the distributed software phone home to access the open source portion of the code from one of your servers. It should also be noted that another way to make money with open source is the professional open source business model pioneered by RedHat; the problem with this kind of model is that a service company must recruit talented people to scale up (a stochastic and non-reproducible process, subject to diminishing returns), while a product company only requires more servers to scale up (which is much more deterministic). As such the profitability and attractiveness of a service company as a VC investment is inherently limited.

6. *OS X client, Linux server*: The most common way to use Linux in Silicon Valley today is in combination with OS X. Specifically, the standard setup is to equip engineers with Macbook Pro laptops for local development and use servers running Linux for production deployment. This is because Apple's OS X runs BSD, a Unix variant which is similar enough to Linux to permit most code to run unchanged on both your local OS X laptop and the remote Linux server. An emerging variant is one in which Chromebooks are used as the local machines with the Chrome SSH client used to connect to a remote Linux development server. This is cheaper from a hardware perspective (Chromebooks

are only $200, while even Macbook Airs are $999) and much more convenient from a system administration perspective (Chromebooks are hard to screw up and easy to wipe), but has the disadvantage that Chromebooks are less useful offline. Still, this variant is worth considering for cash-strapped early stage startups.

7. *Ubuntu: desktop/server Linux distribution*: Perhaps the most popular and all-round polished desktop/server Linux distribution today is Ubuntu, which has a very convenient package management system called apt-get and significant funding for polish from Mark Shuttleworth. You can get quite far by learning Ubuntu and then assuming that the concepts map to other Linux distributions like Fedora/RedHat/CentOS.

8. *Android: mobile Linux distribution*: One last point to keep in mind is that Google's Android OS is also Linux-based and undeniably has many more installed units worldwide than Ubuntu; however, these units are generally smartphones and tablets rather than desktops, and so one usually doesn't think of Android in the same breath as Ubuntu for an engineer's primary development OS (as opposed to a deployment OS used by consumers). That may change, however, as Android on Raspberry Pi continues to gain steam and Pi-sized units become more powerful.

Virtual Machines and the Cloud

The Concept of Virtualization

When you rent a computer from AWS or another infrastructure-as-a-service (IAAS) provider, you are usually not renting a single physical computer, but rather a virtualized piece of a multiprocessor computer. Breakthroughs in operating systems research (in large part from VMWare's founders, both from Stanford) have allowed us to take a single computer with eight processors and make it seem like eight independent computers, each capable of being completely wiped and rebooted without affecting the others. A good analogy for virtualization is sort of like taking a house with eight rooms and converting it into eight independent hotel rooms, each of which has its own 24-hour access and climate control, and can be restored to default settings by a cleaning crew without affecting the other rooms.

Virtualization is extremely popular among corporate IT groups as it allows maximum physical utilization of expensive multiprocessor hardware (which has become the standard due to the flattening of Moore's law on single CPUs). Just like a single person probably won't make full use of an eight-room house, a single 8-CPU server is usually not worked to the limit 24/7. However, an 8-CPU computer split into 8 different virtual servers with different workloads often has much better utilization.

While these "virtual computers" have reasonable performance, they are less than that of the native hardware. And in practice, virtualization is not magic; extremely high workloads on one of the other "tenants" in the VM system can affect your room, in the same way that a very loud next-door neighbor can overwhelm the soundproofing (see the remarks on "Multi-Tenancy" in this blog post). Still, for the many times you don't need an eight-room home or an eight-processor computer, and just need one room/computer for the night, virtualization is very advantageous.

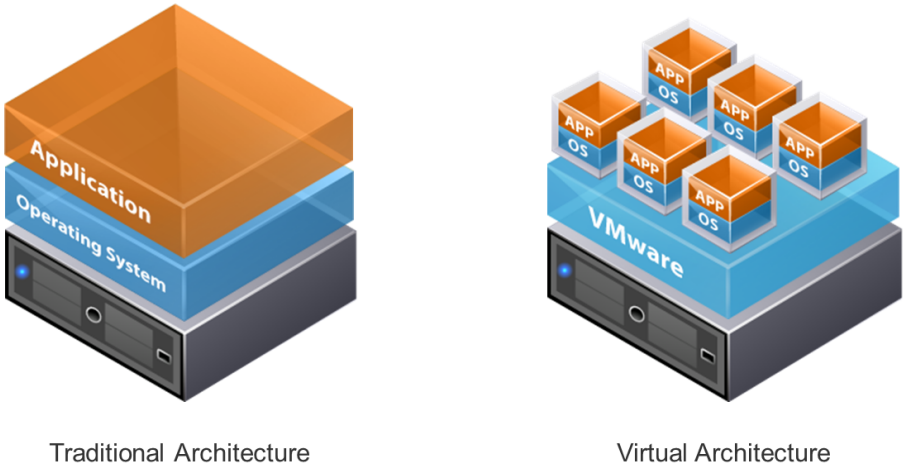

Figure 1: *The concept of virtualization, as illustrated by a figure from VMWare. A single piece of physical hardware can run several different virtual machines, each of which appears to the end user like a full-fledged piece of physical hardware. Individual virtual machines can be shut down or have their (virtual) hard drives reformatted without affecting the uptime of the host OS.*

The Cloud and IAAS/PAAS/SAAS

One definition of a *cloud computer* is a computer whose precise physical location is immaterial to the application. That is, you don't need to have it be present physically to make something happen; it just needs to be accessible over a (possibly wireless) network connection. In practice, you can often determine where a cloud computer is via commands like dig and ping to estimate latency, and there are applications (particularly in real-time communication, like instant messaging, gaming, telephony, or telepresence) where the speed of light and latency correction is a nontrivial consideration (see this famous post).

However, for the large number of applications that can tolerate some (small) degree of latency and/or physical separation from a computer, cloud computing is quite useful. There are three classes of cloud computing, broadly speaking:

1. IAAS: AWS, Joyent, Rackspace Cloud
2. PAAS: Heroku, DotCloud, Nodester, Google AppEngine
3. SAAS: Salesforce, Google Apps, Mint.com

With IAAS you get direct command line access to the hardware, but have to take care of all the details of actually deploying your code. With PAAS by contrast you can drop down to the command line if absolutely necessary, but the platform encourages you to use higher order abstractions instead. The advantage is that this can save you time; the disadvantage is that if you want to do something that the platform authors didn't expect, you need to get back to the hardware layer (and that may not always be feasible). Finally, with SAAS you are interacting solely with an API or GUI and have zero control over the hardware.

Linux Basics

We now have a bit more vocabulary to go through in slow motion what we did during the previous lecture. First, you used an ssh client on your local Windows or Mac to connect to an AWS instance, a virtual machine running Ubuntu 12.04.2 LTS in one of their datacenters. Upon connecting to the instance, you were presented with a bash shell that was waiting for our typed-in commands, much like the Google Search prompt waits for your typed-in searches.

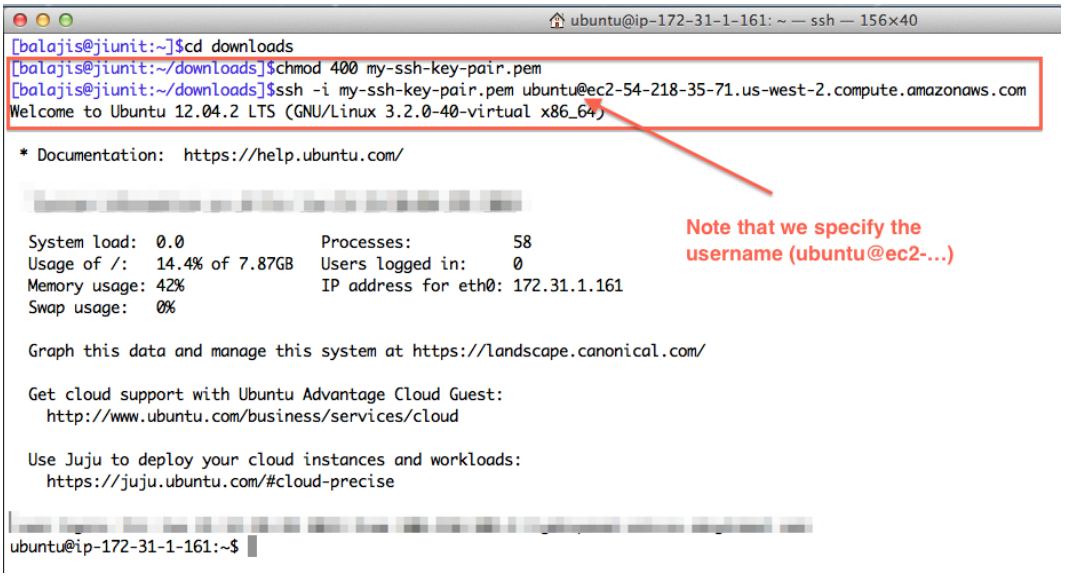

Figure 2: *`ssh` connection to your EC2 instance, pulling up a `bash` shell.*

During the course you will execute many thousands of commands in this environment, so let's first get familiar with the basics of navigating around the filesystem.

Filesystem Basics

You're familiar with the concept of directories and files from working with your local computer. Linux has the same concepts; the main difference is that you generally navigate between directories and create/read/update/delete files by simply typing in commands rather than clicking on files. Let's go through a short interactive example which illustrates how to create and remove files and directories and inspect their properties:

```
# -- Executed on remote machine
cd $HOME  # change to your home directory
pwd  # print working directory
mkdir mydir # make a new directory, mydir
cd mydir
pwd  # now you are in ~/mydir
touch myfile # create a blank file called myfile
ls myfile
ls -alrth myfile # list metadata on myfile
alias ll='ls -alrth' # set up an alias to save typing
ll myfile
echo "line1" >> myfile # append via '>>' to a file
cat myfile
echo "line2" >> myfile
cat myfile
cd ..
pwd
cp mydir/myfile myfile2 # copy file into a new file
cat myfile2
cat mydir/myfile
ls -alrth myfile2 mydir/myfile
rm -i myfile2
cp -av mydir newdir # -av flag 'archives' the directory, copying timestamps
rmdir mydir # won't work because there's a file in there
rm -rf mydir # VERY dangerous command, use with caution
cd newdir
pwd
cp myfile myfile-copy
echo "line3" >> myfile
echo "line4" >> myfile-copy
mv myfile myfile-renamed # mv doubles as a rename
ll
cat myfile-renamed
cat myfile-copy
ll
rm myfile-*
ll
```

```
cd ..
ll
rmdir newdir
ll
```

This should give you an intuitive understanding of how to navigate between directories (`cd`), print the current working directory (`pwd`), print the contents of files (`cat`), list the contents of directories (`ls`), copy files (`cp`), rename files (`mv`), move files (`mv` again), and remove files and directories (`rm`). We'll do much more on this, but you can read more here, here, and here. You might now execute `ls -alrth /` and use `cd` to explore the Ubuntu directory hierarchy.

`env`: List all environmental variables

Just like you can configure parameters like "number of results" which affect what Google Search does as you type in commands, so too can you configure parameters for `bash` called *environmental variables* that will produce different behavior for the same commands. You can see all environmental variables defined in `bash` by running the[1] command `env`:

```
# -- Executed on remote machine
$ env
```

There's an example in Figure 3. We can see for example the current value of `SHELL` is `/bin/bash`, the current value of `USER` is `ubuntu`, and so on. But perhaps the most important is the `PATH`.

[1] `env` can also be used to set up temporary environments for other commands, or in the sha-bang line of a shell script. See here for more.

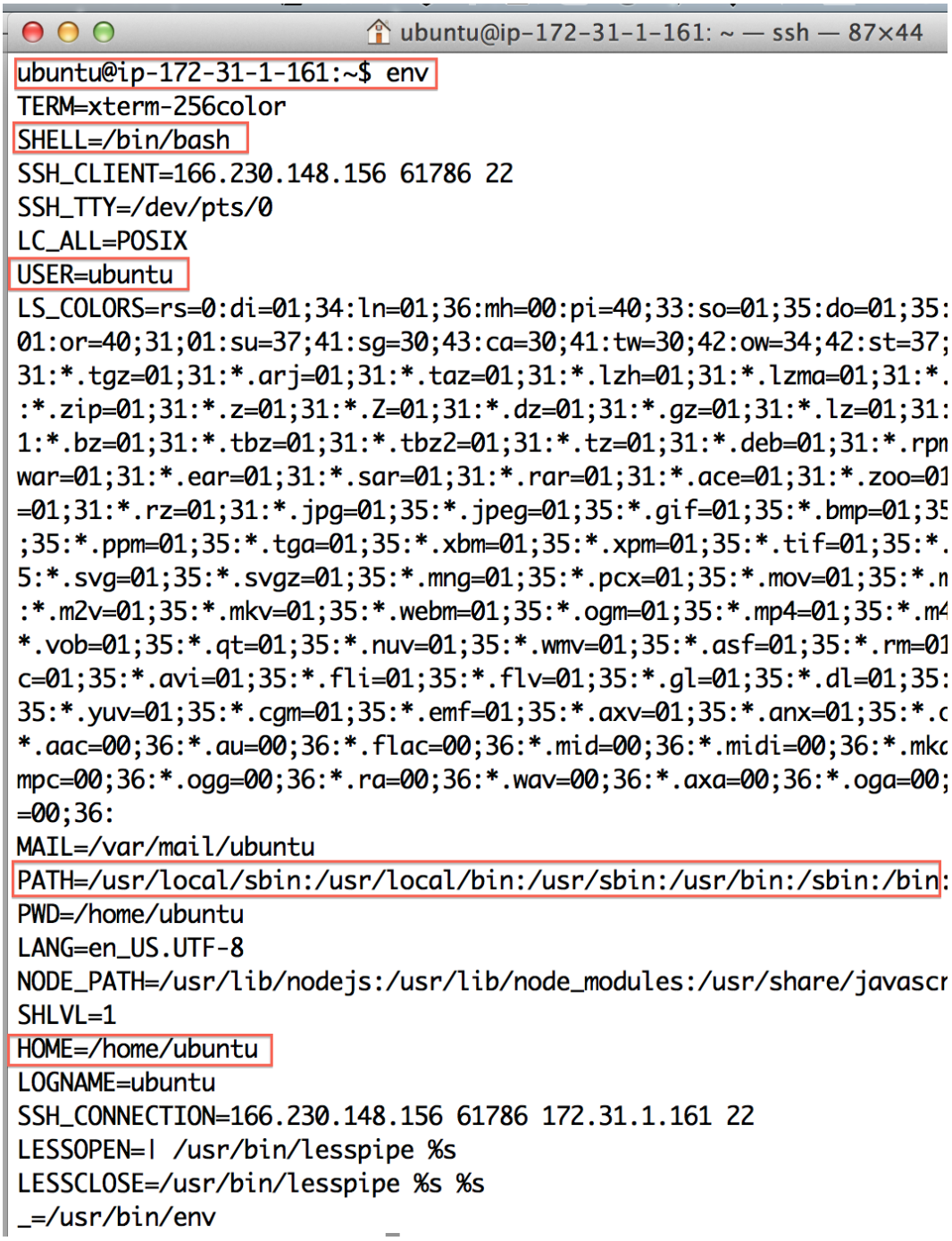

Figure 3: *The* `env` *command prints the* `bash` *environment variables.*

`PATH`: Where Linux looks for programs

Perhaps the most important environmental variable is the `PATH`. It defines the order of directories which the computer will search for programs to execute. This is important when you have installed a program in a non-standard place, or when multiple versions of a program with the same name exist. For example, suppose you have two different versions of `git` installed on your machine in `/usr/bin/git` and `/usr/local/bin/git`. If you don't specify the full pathname on the command line and simply type in `git`, the OS uses the PATH to determine which one you mean. Here are two ways to see[2] the current value of the `PATH`:

```
# -- Executed on remote machine
$ env | grep "^PATH"
$ echo $PATH
```

We see in the case of our stock Ubuntu EC2 12.04.2 LTS instance that the `PATH`'s value is:

```
# -- Executed on remote machine
$ echo $PATH
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

This is a colon-delimited list of directory names. It means the OS will search directories for programs in the following precedence order:

- /usr/local/sbin
- /usr/local/bin
- /usr/sbin
- /usr/bin
- /sbin
- /bin
- /usr/games

You can see this post for more on why this particular order is the default search path for Ubuntu. But the upshot for our purposes is that if `/usr/local/bin/git` and `/usr/bin/git` both existed, then `/usr/local/bin/git` would take precedence because the former directory (`/usr/local/bin`) came before the latter (`/usr/bin`) in the `PATH` (`/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games`). See here for more.

[2]Note that the reason we prepend a `$` to the beginning of `PATH` is that this is how the `bash` shell scripting language identifies variable names (see here for more).

`HOME`: The current user's home directory

The value of another environmental variable also deserves a brief mention. `HOME` is also known as `~` or 'tilde' or the home directory; if you execute `echo $HOME` at the prompt, you will see `/home/ubuntu`, as we are currently logged in as the user named `ubuntu`. You can change to the home directory at any time by typing `cd $HOME`, `cd ~`, or just `cd` and then hitting enter. You can also use `HOME` to make your scripts more portable, rather than hardcoding something like `/home/ubuntu`. By convention, files that are specific to a given user are placed in the home directory. See here for more.

`which`: How to debug `PATH` issues

In some ways the `PATH` is just a convenience, as in theory you could type in the exact full address of every resource on the computer, but local nicknames save a lot of typing. However, `PATH` bugs can be very confusing for newcomers to the command line, and are often the source of subtle bugs[3] even for experienced engineers. The most frequent issue is when you *thought* you installed, upgraded, or recompiled a program, but the `PATH` pulls up an older version or doesn't include your new program. The way to debug this is with the `which` command, as it will tell you where the program you are invoking is located. Try executing the following series of commands:

```
# -- Executed on remote machine
$ which git
$ sudo apt-get install -y git-core
$ which git
```

The first `which git` command may print nothing, if you haven't installed `git` via `apt-get`. The second will print `/usr/bin/git` for certain because you've just installed it. Moreover, you can try executing the following:

```
# -- Executed on remote machine
$ /usr/bin/git --help
$ git --help
```

Why do both of those invocations work? Because `/usr/bin` is in the `PATH`, you can just type in `git` and it will be found. If `git` had been installed in a different location, outside of the `PATH` directories, the second command wouldn't work. So `which` can be used as a basic debugging step to just confirm that the program you are trying to run is even findable in the `PATH`.

What gets more interesting is when there are multiple versions of the same program. For illustrative purposes, let's introduce such a conflict by executing the following commands:

```
# -- Executed on remote machine
$ which -a git
```

[3] This is especially true when compiling C code, as there is not just one `PATH` variable but several such variables for the locations of linker files and the like.

```
/usr/bin/git
$ git
usage: git [--version] [--exec-path[=<path>]] [--html-path] ...
... [snipped]
$ cd $HOME
$ mkdir bin
$ wget https://spark-public.s3.amazonaws.com/startup/code/git
$ mv git bin/
$ chmod u+x bin/git
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
$ export PATH=$HOME/bin:$PATH
$ echo $PATH
/home/ubuntu/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
$ which -a git
/home/ubuntu/bin/git
/usr/bin/git
$ git
fake git
$ /home/ubuntu/bin/git
fake git
$ /usr/bin/git
usage: git [--version] [--exec-path[=<path>]] [--html-path] ...
... [snipped]
$ export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
$ git
usage: git [--version] [--exec-path[=<path>]] [--html-path] ...
... [snipped]
$ /home/ubuntu/bin/git
fake git
$ /usr/bin/git
usage: git [--version] [--exec-path[=<path>]] [--html-path] ...
... [snipped]
```

In this example, we first confirmed that the normal `git` was found in the path (after installation). We then downloaded a second file also named `git`, put it in `~/bin`, and made it executable[4] with `chmod u+x`. We next used the `export` command to redefine the `$PATH` variable, adding `/$HOME/bin` to the front of the path. Then, when we ran `which -a git`, a new entry came to the top: `/home/ubuntu/bin/git`, which is our fake command. And now typing in `git` at the command line just prints out our dummy string, because the precedence order in the `PATH` was changed and `/home/ubuntu/bin/git` came first. Last, we restore the value of the `PATH` and now invoke `git` again. Note finally that no matter what the value of the `PATH`, giving absolute paths to `/home/ubuntu/bin/git` and `/usr/bin/git` always works to specify exactly which `git` you want to execute; the `PATH` is for resolving the shortcut of just typing `git`.

[4] In general, one should be cautious about executing code downloaded from the internet, but in this case this file is from a trusted URL.

`ssh`: Connect to remote machines

The `ssh` command allows you to securely connect to a remote machine. It was a replacement for the old `telnet` command, which sent your password in the clear. Released in 1995, `ssh` really ramped up after the Internet became reasonably popular and security became a concern. Here's some `ssh` commands similar[5] to what we saw in the last lecture:

```
# -- Executed on local machine
# Login to AWS as the ubuntu user using your private key pair
$ cd downloads # or wherever your pem file is
$ ssh -i skey.pem ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com

# Login to AWS as ubuntu using your private key pair and run uptime command
$ ssh -i skey.pem ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com uptime
```

Passing all that information on the command line each time is a little unwieldy, so one way to shorten it is to put some of the information into the `~/.ssh/config` file, as detailed here. When you are done, your config file should look something like this when you `cat` the file[6]:

```
# -- Execute on local machine
$ mkdir -p ~/.ssh
$ cp ~/downloads/skey.pem ~/.ssh/
$ chmod 400 ~/.ssh/skey.pem
$ chmod 700 ~/.ssh
$ nano ~/.ssh/config # edit the file as shown below
$ cat ~/.ssh/config
Host awshost1
HostName ec2-54-218-35-71.us-west-2.compute.amazonaws.com
User ubuntu
IdentityFile "~/.ssh/skey.pem"
```

And then you can simply do this:

```
# -- Execute on local machine
# SSH using the 'awshost1' alias defined in ~/.ssh/config
$ ssh awshost1

# SSH using an alias, run a command, and then exit
$ ssh awshost1 uptime
```

[5] Note that you will need to use your own `pem` files and hostname in order to make this work. You will need to be in the directory with the `.pem` file (probably `~/Downloads` on a Mac) in order to pass it in as a command line argument with `-i`..

[6] In this context, `cat` the file means to print the file to the screen.

The one disadvantage here is that you will need to reenter this information every time you instantiate a new EC2 instance. However, next time[7] you can stop rather than terminate your EC2 instance when you disconnect. This won't guarantee that the host is still up (Amazon can in theory terminate your instances at any time), but it makes it more likely that you won't have to constantly edit this config, and might actually ensure you stay under the free tier hours (see here).

In conclusion, these few commands are most of what you need to know about `ssh`, but as a supplement, here are some further `ssh` examples, and some even more sophisticated things with `ssh` (useful once we learn about pipes).

`scp`: Copy files to/from remote machines.

Just like `ssh` lets you connect to a remote machine[8] and run commands on it, you can use the allied `scp` command to copy files back and forth from the remote machine. This will be very useful for doing homework, as you can edit/generate files on the remote computer and bring them to the local machine, or download datasets for homework assignments through your web browser and then upload them to the remote machine. Here are a few `scp` examples:

```
# -- Execute on local computer
# Copy hello.txt from local computer to remote home directory
$ scp -i skey.pem hello.txt ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com:~/

# Copy h.txt from local to remote home directory, renaming it foo.txt
$ scp -i skey.pem h.txt ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com:~/foo.txt

# Copying ~/foo.txt from remote to current local directory
$ scp -i skey.pem ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com:~/foo.txt .

# Copying ~/foo.txt from remote to local directory cc, renaming it a.b
$ scp -i skey.pem ubuntu@ec2-54-218-73-84.us-west-2.compute.amazonaws.com:~/foo.txt cc/a.b
```

All of these become simpler when you set up your `.ssh/config` file (see this post and the previous section on `ssh`). Then you can do:

```
# -- Execute on local computer
# Copy hello.txt from local computer to remote home directory
$ scp hello.txt awshost1:~/

# Copy hello.txt from local to remote home directory, renaming it foo.txt
$ scp hello.txt awshost1:~/foo.txt
```

[7] One major caveat: given the scale of the class, we might actually want to terminate rather than stop instances to ensure that Amazon has enough capacity. Even for Amazon, 100,000 virtual machines is potentially a big number. They are giving us a free product, so we should be good hosts and not inadvertently abuse that generosity.

[8] Note that it is fairly common to get confused about which box you are currently running commands on, which is why it is common to run the `whoami`, `uname`, or `hostname` commands (or put them in your `$PROMPT` as we will see) to determine which machine you're on. Try them out: they will return different results on your local machine and on your remote EC2 instance.

```
7
8  # Copying ~/foo.txt from the remote computer to the current local directory
9  $ scp awshost1:~/foo.txt .
10
11 # Copying ~/foo.txt from remote to local directory cc, renaming it a.b
12 $ scp awshost1:~/foo.txt cc/a.b
```

Note: all of these `scp` invocations will overwrite the target file if it already exists (aka clobbering), but they will not create a directory if it does not already exist. For that you need the more powerful `rsync` command, which we will cover later. However, you now have the commands necessary to copy homework-related files to and from your local computer to your running EC2 instance.

`bash`: Command interpreter

A command line interface (CLI) is to a search engine as `bash` is to Google. That is, the CLI is the abstract concept and `bash` is one of several specific instantiations called shells. Just like there are many search engines, there are many shells: tcsh, ksh, bash, zsh, and more. `zsh` is superior to (and reverse-compatible with) `bash` in many ways, but what bash has going for it, especially in an introductory class, is ubiquity and standardization. One of the important points about `bash` is that it both has its own built-in commands AND is used to invoke user-installed commands. This is roughly analogous to the way that Google can present both its own results (like a calculator or flight results) or route you to blue links leading to other sites when you type in a command. Here are a few commands to try:

```
1  # -- Execute on EC2 instance
2  # Print all bash environmental variables
3  $ env
4  # Show the value of the HOME variable
5  $ echo $HOME
6  # Change directories to HOME
7  $ cd $HOME
8  # Run a few different ls commands
9  $ ls
10 $ ls *
11 $ ls -alrth *
```

Given how much time you will spend using `bash`, it's important to learn all the keyboard shortcuts. One of the important things about `bash` is that often you will want to execute many commands in a row. You can do this by putting those commands into a single file called a shell script. Often an end user will want some logic in these commands, e.g. to install a package if and only if the operating system is above a particular version. And so it turns out that bash actually includes a full fledged programming language, which is often used to script installs (here's a sophisticated example). We'll use something similar to rapidly configure a new EC2 instance later in the class, but for now here's an example of a simple shell script:

```
#!/bin/bash
date
echo "Good morning, world"
```

The line at the top is called the sha-bang (more here) and is a bit of magic that tells the shell what program to use[9] to interpret the rest of the script.

Unlike the previous code, we don't type the above directly into the prompt. Instead we need to save it as a file. Below are some commands that will accomplish that task; just type them in now, you'll understand more later.

```
# -- Execute on EC2 instance
# Download a remote file
$ wget https://spark-public.s3.amazonaws.com/startup/code/simple.sh
# Print it to confirm you have it; should show a file with "Good morning, world"
$ cat simple.sh
# Check permissions
$ ls -alrth simple.sh
# Set to executable
$ chmod u+x simple.sh
# Confirm that permissions are changed
$ ls -alrth simple.sh
# Execute the file from the current directory
$ ./simple.sh
```

That simple script should print the date (Figure 4).

[9]For example, in this case it's using `/bin/bash`. If instead we typed in `/usr/bin/env node`, it would run the `env` command to find the `node` binary and use that to interpret the rest of the file; this is useful when you might have multiple versions of the node binary.

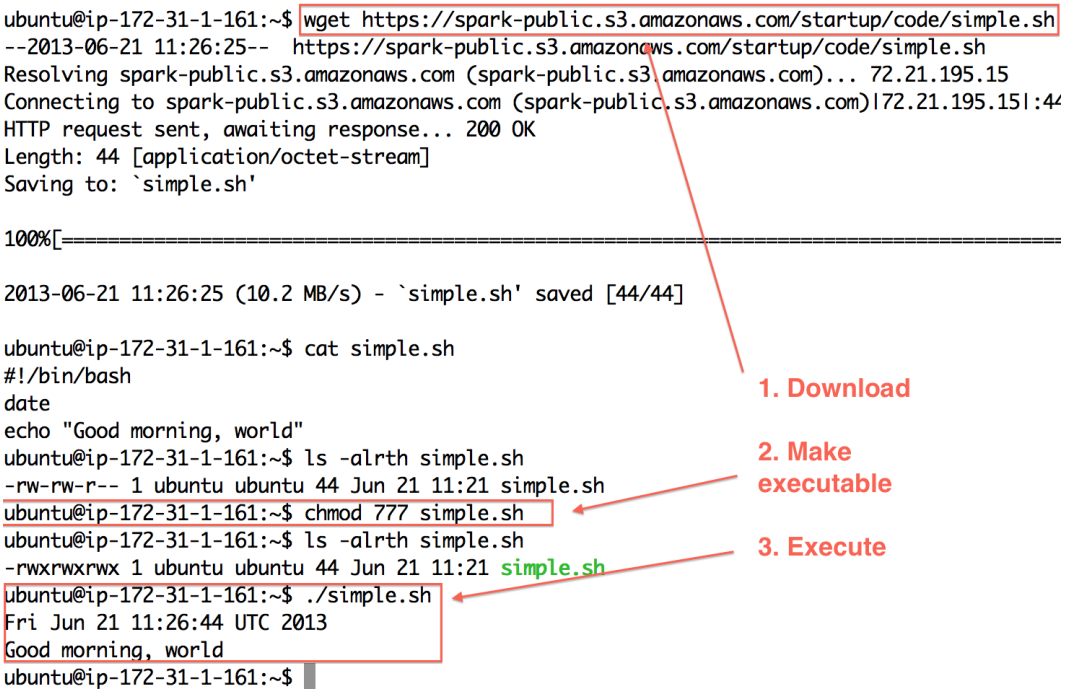

Figure 4: *Simple script that prints the date*

To recap, we just covered `bash` basics: entering commands, `bash` keyboard shortcuts, and the idea of shell scripts. `bash` actually has surprising depth, and you can see more by typing in `man bash` (press q to quit); but let's move on for now.

`apt-get`: Binary package management

There are two major ways to install software in Linux: from *binary packages* and *from source*. Binary packages are pre-compiled packages for your computer's architecture that can be installed as rapidly as they are downloaded. Here is a quick example of installing `git` via Ubuntu's binary package manager, `apt-get`:

```
# -- Execute on remote machine
$ which git
$ sudo apt-get install -y git-core
# ... lots of output ...
$ which git
/usr/bin/git
```

That was fast and easy. If we want to uninstall, we just need to type in `sudo apt-get remove git-core`, and you can learn more apt-get commands here. There are disadvantages to binary packages, though, including:

- because they are pre-compiled, you will not be able to customize binaries with compile-time flags that change behavior in fundamental ways, which is especially important for programs like nginx.
- only older versions of packages will usually be available in binary form.
- newer or obscure software may not be available at all in binary form.
- they may be a bit slower or not 100% optimized for your architecture, especially if your architecture is odd or custom; often this is related to the compile-time flag issue.

That said, for most non-critical software you will want to use binary packages for speed and compatibility.

Compiling from source: `./configure; make; make install`

In general, one wants to use `apt-get` or similar binary package managers as much as possible. Someone else has gone through the hassle of looking through the dependencies of each package and giving you a precooked binary that you can just download and run without worrying about it. Binary packages are the easy way. However, for recent or mission-critical software, like `nginx` or PostgreSQL, you will sometimes want to install packages the hard way: compile from source. Here is an example of doing this for the sqlite lightweight database

```
# First we install the C compiler gcc and the build tool make
$ sudo apt-get update
$ sudo apt-get install -y make gcc
```

```
4
5  # Next we download the .tar.gz
6  $ wget http://www.sqlite.org/2013/sqlite-autoconf-3071700.tar.gz
7
8  # We untar it and gunzip it at the same time
9  $ tar -xzvf sqlite-autoconf-3071700
10 $ cd sqlite-autoconf-3071700
11
12 # Then we build and install it globally.
13 # NOTE: this is NOT advised under most circumstances; see below.
14 $ ./configure
15 $ make
16 $ sudo make install
17
18 # Finally, we tell the system that new libraries are available
19 $ sudo ldconfig -v
20
21 # And we try it out
22 $ /usr/local/bin/sqlite3 --version
23 3.7.17 2013-05-20 00:56:22 118a3b35693b134d56ebd780123b7fd6f1497668
```

While the set of three commands (`./configure; make; sudo make install`) is canonical, in the sense that you will see it all over the place, this common procedure is rather unsafe. The reason is that because it is run as root and because it writes to global directories (`/usr/bin/local`), this combination of commands can (and usually will) overwrite existing programs, possibly used by other parties. And unlike `apt-get remove`, it also provides no easy means for uninstall. While we can easily throw away and rebuild any given EC2 instance, it would be much more of a pain to do so on your local Mac if things go wrong. So you normally want to do something like this instead:

```
1  $ ./configure --prefix=$HOME/my-sqlite-dir # pick whatever name you want
2  $ make
3  $ make install
```

That will force the entire program to build and install in `~/my-sqlite-dir`, a fresh/new directory under your `HOME`, making it much easier to delete when necessary, as shown:

```
1  ubuntu@ip-172-31-17-82:~/my-sqlite-dir$ ls -alrth
2  total 24
3  drwxrwxr-x 6 ubuntu ubuntu 4096 Jan 23 01:30 ./
4  drwxr-xr-x 6 ubuntu ubuntu 4096 Jan 23 01:30 ../
5  drwxrwxr-x 2 ubuntu ubuntu 4096 Jan 23 01:30 bin/
6  drwxrwxr-x 2 ubuntu ubuntu 4096 Jan 23 01:30 include/
7  drwxrwxr-x 3 ubuntu ubuntu 4096 Jan 23 01:30 lib/
8  drwxrwxr-x 3 ubuntu ubuntu 4096 Jan 23 01:30 share/
9
```

```
ubuntu@ip-172-31-17-82:~/my-sqlite-dir$ cd bin
ubuntu@ip-172-31-17-82:~/my-sqlite-dir/bin$ ll
total 164
drwxrwxr-x 2 ubuntu ubuntu   4096 Jan 23 01:30 ./
drwxrwxr-x 6 ubuntu ubuntu   4096 Jan 23 01:30 ../
-rwxr-xr-x 1 ubuntu ubuntu 155801 Jan 23 01:30 sqlite3*

ubuntu@ip-172-31-17-82:~/my-sqlite-dir$ cd $HOME
ubuntu@ip-172-31-17-82:~/$ mkdir bin; cd bin
ubuntu@ip-172-31-17-82:~/bin$ ln -s ../my-sqlite-dir/bin/sqlite3 .
ubuntu@ip-172-31-17-82:~/bin$ cd
ubuntu@ip-172-31-17-82:~$ export PATH=$HOME/bin:$PATH
ubuntu@ip-172-31-17-82:~$ which -a sqlite3
ubuntu@ip-172-31-17-82:~$ sqlite3 --version
3.7.17 2013-05-20 00:56:22 118a3b35693b134d56ebd780123b7fd6f1497668
/home/ubuntu/bin/sqlite3
```

Installing into your own personal directory solves three problems:

- You can easily just do `rm -rf ~/my-sqlite-dir` to uninstall, as you know that every file was written to a subdirectory of `~/my-sqlite-dir`
- You can link the corresponding binaries it into your `$PATH`, e.g. by symbolically linking them into `$HOME/bin` with `ln -s` as shown.
- You avoid conflicts with other versions of the code, or other people's versions of the code.

In short: `./configure --prefix=$HOME/mydir; make; make install` is the preferred way to safely install things from source.

Example of installation conflicts (optional)

This section is optional, as it will mess up your EC2 instance if you execute these commands. Still, this can be handy as it will allow you to recognize this or similar issues should they recur inadvertently in the future. Without further ado, suppose that after installing `sqlite3` from source with `./configure; make; sudo make install` we had then run this command to install another version of `sqlite3` from `apt-get`:

```
$ sudo apt-get install -y sqlite3
```

Now we have a problem. Try executing `sqlite3 --version` or `/usr/bin/sqlite3 --version`, and you will likely get a message like this:

```
ubuntu@ip-10-110-59-51:~$ /usr/bin/sqlite3 --version
SQLite header and source version mismatch
2013-05-20 00:56:22 118a3b35693b134d56ebd780123b7fd6f1497668
2011-11-01 00:52:41 c7c6050ef060877ebe77b41d959e9df13f8c9b5e
```

If we are more specific and invoke with the full path to each program then we get this:

```
ubuntu@ip-10-110-59-51:~$ /usr/bin/sqlite3 --version
SQLite header and source version mismatch
2013-05-20 00:56:22 118a3b35693b134d56ebd780123b7fd6f1497668
2011-11-01 00:52:41 c7c6050ef060877ebe77b41d959e9df13f8c9b5e

ubuntu@ip-10-110-59-51:~$ /usr/local/bin/sqlite3 --version
3.7.17 2013-05-20 00:56:22 118a3b35693b134d56ebd780123b7fd6f1497668
```

What's going on here? In short, by installing a package from source via `./configure; make; make install` AND via `apt-get` we confused the system. Invoking `/usr/bin/sqlite3` calls the version installed via `apt-get`, but there are now other paths for headers/libraries that aren't in sync. Sometimes `sudo apt-get remove sqlite3` or the equivalent will fix the problem, but not always (especially if you've overwritten files in `/usr/bin/` rather than just `/usr/local/bin`). So the lesson here is that you can easily screw things up if you

- install to `/usr/local` with `./configure; make; sudo make install`
- install both binary and compiled packages

So don't do either of those things. Try to stick with binary packages and use `./configure --prefix=$HOME` otherwise. If you must install from source, install to a directory in your home by doing `./configure --prefix=$HOME/mydir; make; make install` rather than globally installing outside your `HOME` directory with `./configure; make; sudo make install`.

Linux Recap and Summary

All right. So now we can understand how to initialize a virtual machine, connect to it via ssh, launch a bash shell, and install some software via binary packages to configure our machine. We also can move around and create files, and perhaps know a bit about the Ubuntu directory hierarchy. When combined with the previous lectures, we can now dial up a Linux machine in the cloud, configure it to a fair extent, and start executing code! Awesome progress. Now let's write a little code.

Basic Server-Side JS (SSJS)

Install `node` and `npm`

We're going to start in this first lesson by treating `node` as just another scripting language like `python` or `perl`, without any of the special features of node (networking, async invocation, shared client-server code). Let's begin by executing the following commands on an EC2 instance to set up `node` and `npm` (the node package manager).

```
$ sudo apt-get update
# Install a special package
$ sudo apt-get install -y python-software-properties python g++ make
# Add a new repository for apt-get to search
$ sudo add-apt-repository ppa:chris-lea/node.js
# Update apt-get's knowledge of which packages are where
$ sudo apt-get update
# Now install nodejs and npm
$ sudo apt-get install -y nodejs
```

And now you have the `node` and `npm` commands. You can check that these exist by invoking them as follows:

```
$ npm --version
1.2.32
$ node --version
v0.10.12
```

The `node.js` REPL

A REPL is a read-evaluate-print loop. It refers to an environment where you can type in commands in a programming language, one line at a time, and then immediately see them execute. You can launch the `node` REPL at the prompt by simply typing `node`:

```
[ubuntu@ip-10-204-245-82:~]$node
> 1 + 1
2
> var foo = "asdf";
undefined
> foo
'asdf'
> .help
.break  Sometimes you get stuck, this gets you out
.clear  Alias for .break
.exit   Exit the repl
.help   Show repl options
.load   Load JS from a file into the REPL session
.save   Save all evaluated commands in this REPL session to a file
>
```

Figure 5: *The node.js Read-Evaluate-Print-Loop (REPL) is launched by simply typing* `node` *at the command line after installing it.*

Editing code with `nano`

We'll use the `nano` code editor for your first few programs; using something primitive/simple like this will increase your appreciation for the combination of `emacs` and `screen`. As shown in Figure 6, it is useful to open up two separate connections to your remote machine (e.g. two Terminal.app windows on a Mac, or two Cygwin windows on Windows):

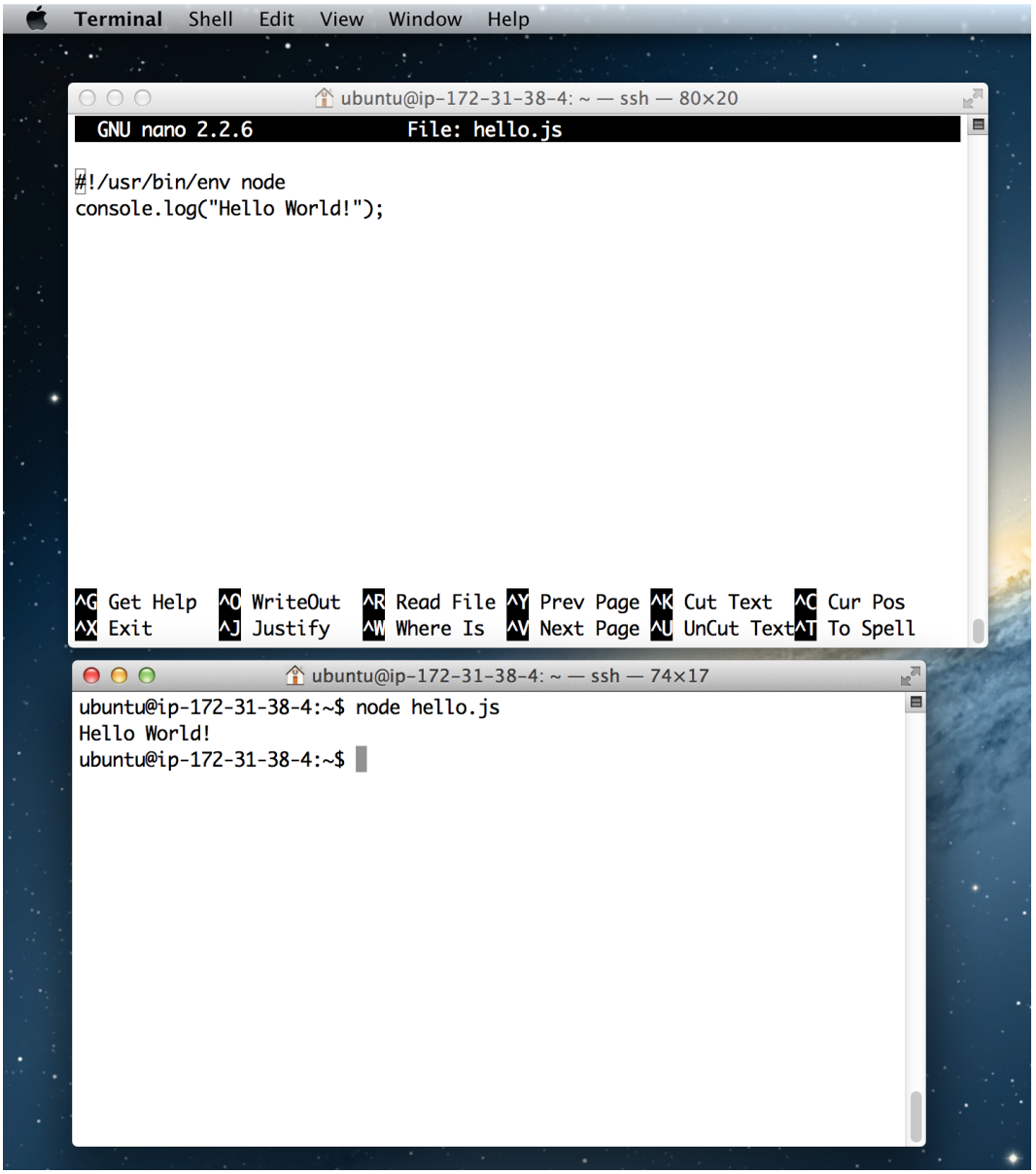

Figure 6: *Here's an example of editing with two windows open and connected to the remote host, one for the* `nano` *editor and one with the main* `bash` *prompt. We'll see a much better way to do this soon with GNU screen.*

A first `node.js` program: Hello World

If you want to zip ahead with node, you can't go wrong by going through the Node.js Beginner's Book. But let's do our first `node.js` program, just to confirm it works. Open a file named `hello.js` with the `nano` editor by typing `nano hello.js`. Type in `#!/usr/bin/env node`, hit enter, type in `console.log("Hello World");` and then save and quit (by hitting Control-X, then yes, then Enter). Then execute the file with `node hello.js`. Here is how that should look:

```
[ubuntu@ip-10-204-245-82:~/nodedemo]$nano hello.js  # edit as described above
[ubuntu@ip-10-204-245-82:~/nodedemo]$cat hello.js
#!/usr/bin/env node
console.log("Hello World");
[ubuntu@ip-10-204-245-82:~/nodedemo]$node hello.js
Hello World
```

If this works, great job. You've got a working interpreter and can do the first part of the HW1 programming assignment.

A second `node.js` program: Using the libraries

Now try typing in the following script with `nano` in a file named `hello2.js`:

```
#!/usr/bin/env node
var fs = require('fs');
var outfile = "hello.txt";
var out = "Modify this script to write out something different.\n";
fs.writeFileSync(outfile, out);
console.log("Script: " + __filename + "\nWrote: " + out + "To: " + outfile);
```

The two new wrinkles of note are that we are using the `writeFileSync` function from the built-in filesystem (`fs`) library (more details if you want, but don't worry about them yet) and are also using the built-in node.js convention that `__filename` is a variable with the path to the current file. Here is what it looks like if we print the file to the screen with `cat` and then execute it:

```
ubuntu@ip-172-31-1-161:~$ cat hello2.js
#!/usr/bin/env node
var fs = require('fs');
var outfile = "hello.txt";
var out = "Modify this script to write out something different.\n";
fs.writeFileSync(outfile, out);
console.log("Script: " + __filename + "\nWrote: " + out + "To: " + outfile);

ubuntu@ip-172-31-1-161:~$ node hello2.js
Script: /home/ubuntu/hello2.js
Wrote: A startup is a business built to grow rapidly.
```

```
To: hello.txt

ubuntu@ip-172-31-1-161:~$ cat hello.txt
A startup is a business built to grow rapidly.
```

As shown above, if you `cat hello.txt` it has the string we expected. Note also that you can use `chmod` to make the `hello2.js` script executable, as follows:

```
ubuntu@ip-172-31-1-161:~$ chmod u+x hello2.js

ubuntu@ip-172-31-1-161:~$ ./hello2.js
Script: /home/ubuntu/hello2.js
Wrote: Modify this script to write out something different.
To: hello.txt
```

In this example `bash` is using the sha-bang `#!/usr/bin/env node` in the first line of `hello2.js` to determine what program to use to interpret the script. Don't worry about it if you don't get it fully yet - for now, just remember to include that sha-bang line (`#!/usr/bin/env node`) at the top of any `node` server-side script, and then do `chmod u+x` to make it executable.

A third `node.js` program: Fibonacci

Here's our third node program. Do `nano fibonacci.js` and type in[10] the following code. You can of course copy it locally and `scp` it to the remote machine, but often typing in code yourself is a good exercise.

```
#!/usr/bin/env node

// Fibonacci
// http://en.wikipedia.org/wiki/Fibonacci_number
var fibonacci = function(n) {
    if(n < 1) { return 0;}
    else if(n == 1 || n == 2) { return 1;}
    else if(n > 2) { return fibonacci(n - 1) + fibonacci(n - 2);}
};

// Fibonacci: closed form expression
// http://en.wikipedia.org/wiki/Golden_ratio#Relationship_to_Fibonacci_sequence
var fibonacci2 = function(n) {
    var phi = (1 + Math.sqrt(5))/2;
    return Math.round((Math.pow(phi, n) - Math.pow(1-phi, n))/Math.sqrt(5));
};

// Find first K Fibonacci numbers via basic for loop
```

[10] OK, if you are really lazy, just do this: `wget https://spark-public.s3.amazonaws.com/startup/code/fibonacci.js`

```
var firstkfib = function(k) {
    var i = 1;
    var arr = [];
    for(i = 1; i < k+1; i++) {
        arr.push(fibonacci(i));
    }
    return arr;
};

// Print to console
var fmt = function(arr) {
    return arr.join(" ");
};

var k = 20;
console.log("firstkfib(" + k + ")");
console.log(fmt(firstkfib(k)));
```

In general, if you have a CS106B level background as listed in the prerequisites, this should be a fairly straightforward bit of code to understand. It uses basic programming constructs like variables, if/else statements, for loops, functions, and the like.

The only things that might not be completely obvious are the use of the `push` and `join` methods from the `Array` object (see here), and the use of the `round`, `pow`, and `sqrt` methods from the `Math` object (see here). The function declaration syntax might look a little weird too (in the sense of assigning a function to a named variable), but you can just accept it for now as a preview of what is to come (1, 2, 3). Executing this script should give you the first 20 Fibonacci numbers:

```
[balajis@jiunit:~]$chmod u+x fibonacci.js
[balajis@jiunit:~]$./fibonacci.js
firstkfib(20)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

Once you understand this short script, you should be able to adapt it to solve the prime number calculation problem in the homework.