

Linux

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 expose 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 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 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; it is ramping in popularity with major projects like [MongoDB](#) adopting 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 requires recruiting talented people to scale (a stochastic and non-reproducible process, subject to diminishing returns), while a product company requires simply [racking more servers](#) (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 conjunction 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: The most popular Linux distribution today is [Ubuntu](#), which has a very convenient package management system called [apt-get](#) and significant funding for polish. You can get quite far by learning Ubuntu and then assuming that the concepts map to other Linux distributions like [Fedora/RedHat/CentOS](#).

Virtual Machines and the Cloud

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 Stanford professors) 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 as it allows maximum physical utilization of expensive multiprocessor hardware. 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.

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](#)

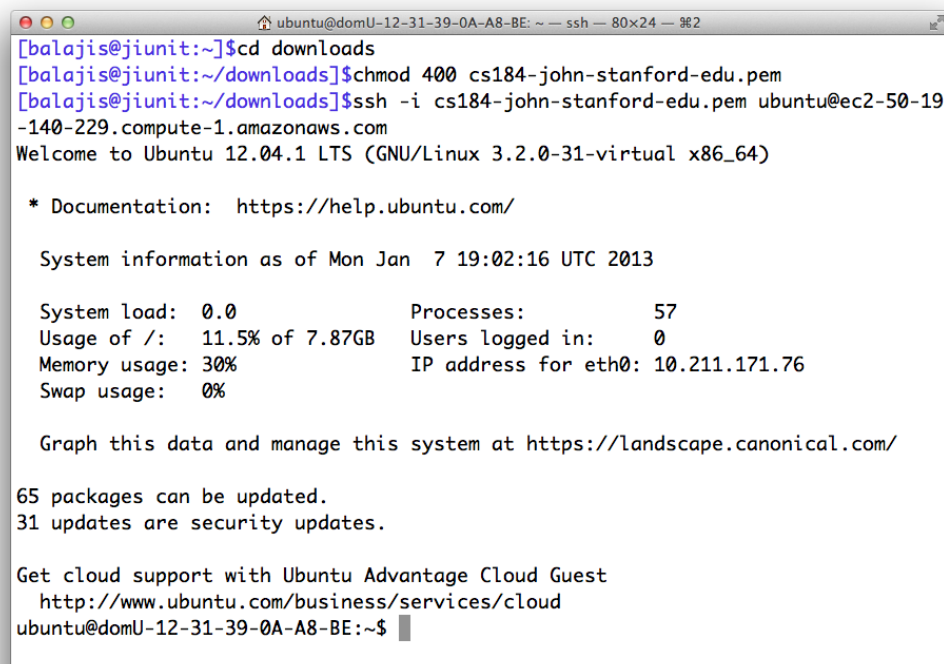
2. PAAS: Heroku, DotCloud, Nodester, Google AppEngine

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 last lecture. First, you used an ssh client on your local Windows or Mac to connect to an AWS instance, a virtual machine running Ubuntu 12.04.1 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.



```
ubuntu@domU-12-31-39-0A-A8-BE: ~ — ssh — 80x24 — 962
[balajis@jiunit:~]$ cd downloads
[balajis@jiunit:~/downloads]$ chmod 400 cs184-john-stanford-edu.pem
[balajis@jiunit:~/downloads]$ ssh -i cs184-john-stanford-edu.pem ubuntu@ec2-50-19-140-229.compute-1.amazonaws.com
Welcome to Ubuntu 12.04.1 LTS (GNU/Linux 3.2.0-31-virtual x86_64)

* Documentation:  https://help.ubuntu.com/

System information as of Mon Jan  7 19:02:16 UTC 2013

System load:  0.0               Processes:            57
Usage of /:   11.5% of 7.87GB    Users logged in:     0
Memory usage: 30%              IP address for eth0: 10.211.171.76
Swap usage:   0%

Graph this data and manage this system at https://landscape.canonical.com/

65 packages can be updated.
31 updates are security updates.

Get cloud support with Ubuntu Advantage Cloud Guest
http://www.ubuntu.com/business/services/cloud
ubuntu@domU-12-31-39-0A-A8-BE:~$
```

Figure 1: *ssh connection to your EC2 instance, pulling up a bash shell.*

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. In addition, you can use the **bash** shell to install new software with just a few keystrokes. Over the course of the quarter you will execute many thousands of commands in this environment, so let’s cover each of these in a little more detail.

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; surprisingly, **ssh** only really ramped up after the Internet became reasonably popular, as people were using **telnet** well into the late 90s. Here are some examples of **ssh** usage:

```
1 # Login as the current user
2 $ ssh elaine.stanford.edu
3
4 # Login as a specific user
5 $ ssh john@elaine.stanford.edu
6
7 # Run the uptime command on elaine.stanford.edu
8 $ ssh john@elaine.stanford.edu uptime
```

Note that by executing these on the EC2 box we are doing two layers of ssh connections. It is fairly common to get confused about which box you are on, why it is common to run **whoami** and **hostname** (or put them in your **\$PROMPT** as we will see) to determine which machine you’re on. You can also use **ssh** to run commands from your *local* terminal on the EC2 instance:

```
1 [john@localhost:/home/john]$ ssh -i .ssh/cs184-john-stanford-edu.pem \
2 ubuntu@ec2-50-17-12-211.compute-1.amazonaws.com date
3 Tue Jan 15 15:24:16 UTC 2013
```

Try all those commands now (with your own username rather than **john**). If you have an active Stanford ID you should be able to log in to **elaine.stanford.edu** with your Stanford password. Later in the class we’ll do some more sophisticated things with **ssh**, once we learn about Unix pipes.

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

```

Given how much time you will spend using bash, it's important to learn all the keyboard shortcuts. Watch this [screencast](#), noting that `^A` means to hold down the Control key while pressing A, and this [symbol](#) means to hold down Option while pressing A.

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](#):

```

1 #!/bin/bash
2 clear
3 date
4 echo "Good morning, world"

```

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.

```

1 # Download a remote file
2 $ wget http://startup-class.s3.amazonaws.com/simple.sh
3 # Print it to confirm
4 $ cat simple.sh
5 # Check permissions
6 $ ls -alrth simple.sh
7 # Set to executable
8 $ chmod 777 simple.sh
9 # Confirm that permissions are changed
10 $ ls -alrth simple.sh
11 # Execute the file from the current directory
12 $ ./simple.sh

```

All right. This covers 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.

screen: Tab manager for remote sessions

When Google Chrome or Firefox crashes, it offers to restore your tabs upon reboot. The screen program is similar: when your network connection to the remote host drops out, you can reconnect and resume your session. This is more than a convenience; for any long-running computation, screen (or background processes) is the only way to do anything nontrivial. For the purposes of this class, let's use a custom `.screenrc` file which configures `screen` to play well with the `emacs` text editor (to be introduced shortly). You can install this config file as follows:

```
1 $ cd $HOME
2 $ wget http://startup-class.s3.amazonaws.com/.screenrc
3 $ head .screenrc
4 $ screen
```

Let's go through an interactive session to see the basics of `screen`. There's an excellent tutorial and video tutorial if you want to get into more detail.

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 node.js via Ubuntu's binary package manager, apt-get:

```
1 # Install a special package
2 $ sudo apt-get install -y python-software-properties
3 # Add a new repository for apt-get to search
4 $ sudo add-apt-repository ppa:chris-lea/node.js
5 # Update apt-get's knowledge of which packages are where
6 $ sudo apt-get update
7 # Now install nodejs and npm
8 $ sudo apt-get install -y nodejs npm
```

And now you have the node and npm commands, which will be handy later:

```
1 $ npm --version
2 1.1.71
3 $ node --version
4 v0.8.17
```

That was fast and easy. 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.

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

What about your most critical software, like [nginx](#) or PostgreSQL? If you need maximum control, you will want to compile from source. Here is an example of doing this for the [sqlite](#) lightweight database:

```
1 $ sudo apt-get install -y make
2 $ wget http://www.sqlite.org/sqlite-autoconf-3071502.tar.gz
3 $ tar -xzf sqlite-autoconf-3071502.tar.gz
4 $ cd sqlite-autoconf-3071502
5 $ ./configure
6 $ make
7 $ sudo make install
8 $ ldconfig -v
```

 While canonical, this procedure is rather unsafe as it can overwrite existing programs (possibly used by other parties). It also provides no easy means for uninstall. While we can easily throw away and rebuild this AMI, 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
2 $ make
3 $ make install
```

That will force the entire program to build and install under your \$HOME directory, making it much easier to delete when necessary. You can then link it into your \$PATH.


\$PATH: Where Unix looks for programs

You can see all environmental variables defined in `bash` by running the command `env`:

```
1 $ env
```


Perhaps the most important environmental variable is the PATH. It defines the search order in which the computer will look 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. Here are two ways to see the current value of the PATH:

```
1 $ env | grep "^PATH"
2 $ echo $PATH
```



PATH bugs can be very confusing for newcomers to the command line, and are often the source of subtle bugs 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:

```
1 $ which heroku
2 /usr/bin/heroku
3 $ which sqlite3
4 /usr/local/bin/sqlite3
5 $ which node
6 /usr/bin/node
7 $ which npm
8 /usr/bin/npm
9 $ which foobarbaz
```

The last command here will find nothing, as there is no program of the name **foobarbaz**. What gets more interesting is when there are multiple versions of the same program. Just for kicks, let's introduce such a conflict:

```
1 $ which -a sqlite3
2 /usr/local/bin/sqlite3
3 $ sudo apt-get install -y sqlite3
4 $ which -a sqlite3
5 /usr/local/bin/sqlite3
6 /usr/bin/sqlite3
7 $ which sqlite3
8 /usr/local/bin/sqlite3
```

Interesting. Invoking **which** with the **-a** flag shows ALL commands that are aliased to that name. Before we installed sqlite3 via apt-get, we only picked up one version of sqlite3. Afterwards, we had two, in **/usr/local/bin/sqlite3** and **/usr/bin/sqlite3**. Why does the **/usr/local/bin/sqlite3** come first, though? Let's look at the **\$PATH** again:

```
1 $ echo $PATH | sed 's:/:\n/g'
2 /usr/local/heroku/bin
3 /usr/local/sbin
```

```
4 /usr/local/bin
5 /usr/sbin
6 /usr/bin
7 /sbin
8 /bin
9 /usr/games
```

We see that /usr/local/bin comes before /usr/bin. We can switch this around as follows:

```
1 $ which sqlite3
2 /usr/local/bin/sqlite3
3 $ export PATH=/usr/bin:$PATH
4 $ echo $PATH | sed 's:/:\n/g'
5 /usr/bin
6 /usr/local/heroku/bin
7 /usr/local/sbin
8 /usr/local/bin
9 /usr/sbin
10 /usr/bin
11 /sbin
12 /bin
13 /usr/games
14 $ which sqlite3
15 /usr/bin/sqlite3
```

A word of caution This section is optional. If you try executing `sqlite3 --version` you will likely get a message like this:

```
1 ubuntu@ip-10-110-59-51:~$ sqlite3 --version
2 SQLite header and source version mismatch
3 2013-01-09 11:53:05 c0e09560d26f0a6456be9dd3447f5311eb4f238f
4 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:

```
1 ubuntu@ip-10-110-59-51:~$ /usr/bin/sqlite3 --version
2 SQLite header and source version mismatch
3 2013-01-09 11:53:05 c0e09560d26f0a6456be9dd3447f5311eb4f238f
4 2011-11-01 00:52:41 c7c6050ef060877ebe77b41d959e9df13f8c9b5e
5
6 ubuntu@ip-10-110-59-51:~$ /usr/local/bin/sqlite3 --version
7 3.7.15.2 2013-01-09 11:53:05 c0e09560d26f0a6456be9dd3447f5311eb4f238f
```



What's going on here? In short, by installing a package from source AND via apt-get we confused the system. The \$PATH now points to the version installed via apt-get, but there are other paths for headers/libraries that aren't moving in sync. We're not going to debug this right now, but 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 that. Try to stick with binary packages and use ./configure --prefix=\$HOME otherwise.

An Interactive Introduction

We're now going to go through a short interactive example which illustrates how to create and remove files and directories and inspect their properties. Execute these commands on your remote EC2 instance as always.

```
1 cd $HOME # change to home
2 pwd # print working directory
3 mkdir mydir # make a new directory, mydir
4 cd mydir
5 touch myfile # create a blank file called myfile
6 ls myfile
7 ls -alrth myfile
8 alias ll='ls -alrth' # set up an alias to save typing
9 ll myfile
10 echo "line1" >> myfile # append via '>>' to a file
11 cat myfile
12 echo "line2" >> myfile
13 cat myfile
14 cd ..
15 cp -av mydir newdir # -av flag 'archives' the directory, copying timestamps
16 rmdir mydir # won't work because there's a file in there
17 rm -rf mydir # VERY dangerous command, use with caution
18 cd newdir
19 pwd
20 cp myfile myfile-copy
21 echo "line3" >> myfile
22 echo "line4" >> myfile-copy
23 mv myfile myfile-renamed # mv doubles as a rename
24 ll
25 cat myfile-renamed
26 cat myfile-copy
27 ll
28 rm myfile-*
29 ll
30 cd ..
```

```
31 | ll
32 | rmdir newdir
33 | ll
```

Exploring Ubuntu

We now know how to navigate between directories, and create/rename/move/copy files. As a final step for today, execute `ls -alrth /` and use `cd` to explore the [Ubuntu directory hierarchy](#).

Summary

All right. So now we can understand how to initialize a virtual machine, connect to it via ssh, launch a bash shell, get a screen instance running on the remote machine so as not to lose our work in the event of a disconnection, install some software (via binary packages or from source) to configure our machine, and find/configure that software in our `$PATH`. We also can move around and create files, and 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 and execute commands on it! Awesome progress.

Miscellaneous: OS X Tip

One note: if you are on a Mac, it will make your life easier to set up your Keyboard Preferences in the following way:



Figure 2: *Set your repeat rates to extremely fast.*

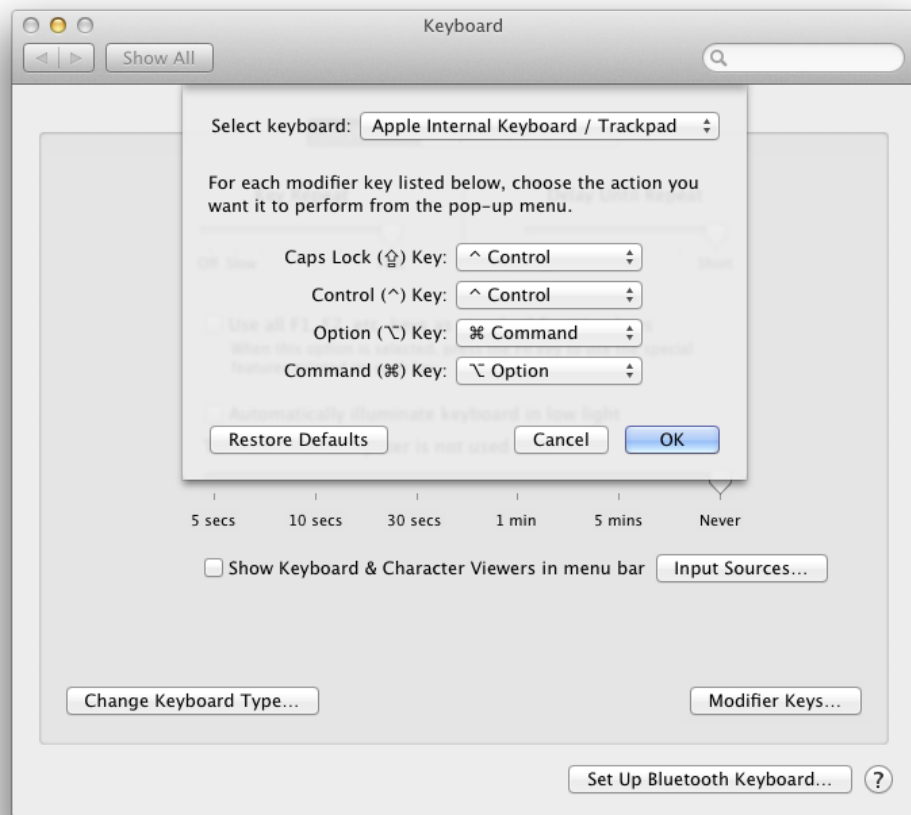


Figure 3: *Switch Caps-lock and Control, and (at your discretion) switch Option and Command. The latter will make emacs much easier to use.*