Introduction to Linux

UNIX Basics

UNIX is a multi-tasking, multi-user operating system, which is well-suited to programming and programming-related tasks (running servers, etc.). Technically speaking, UNIX refers to a specific operating system developed at Bell Labs in the 1970s; however, today it is more commonly used (slightly imprecisely) to mean "any UNIX-like" operating system, such as Linux, Free BSD, Solaris, AIX, and even Mac OSX. Here, we will use the more general term, and note that you are most likely to use Linux or Mac OSX.

UNIX is a great example of the tools for experts versus tools for novices tradeoffs discussed earlier in this course. If you are reading this section, odds are good that you fall into the relatively large set of people who are "master novices" when it comes to using a computer—that is, you have mastered all of the skills of a novice system. You can use a graphical interface to open files, send email, browse the web, and play music. Maybe you can even fix a few things when something goes wrong. However, you would be hard pressed to make your computer perform moderately sophisticated tasks in an automated fashion.

As a simple example, suppose you had 50 files in a directory (aka "folder") and wanted to rename them all by replacing _ with - in their names (but otherwise leaving the names unchanged). As a "master novice" you could perform this task in the graphical interface by hand—clicking each file, clicking rename, and typing in the new name. However, such an approach would be incredibly tedious and time consuming. An expert user would use the command line (which we will introduce shortly) to rename all 50 files in a single command, taking only a few seconds of work.

In the Beginning Was the Command Line

While UNIX has a graphical user interface (GUI), its users often make use of the *command line*. In its simplest usage, the command line has you type the name of the program you want to run, whereas a GUI-based operating system might have you double-click on an icon of the program you want to run. The command line interface can be intimidating or frustrating at first, but an expert user will often prefer the command line to a GUI. Beyond being the natural environment to program in, it allows for us to perform more sophisticated tasks, especially automating those which might otherwise be repetitive.

To reach a command line prompt, you will need to use a terminal emulator (commonly referred to as just a "terminal"), which is a program that emulates a text-mode terminal. In this course, you can do everything you need to do on our Mastery Learning Platform. We'll tell you soon how to login to that and do your classwork there. However, if you ever have the need outside of this class, you can also access a terminal on your machine. If you are running a UNIX based system (Linux or Mac OSX), a terminal is available natively. In Linux, if you are using the graphical environment, you can run xterm, or you can switch to an actual text-mode terminal by pressing Ctrl-Alt-F1 (to switch back to the graphical interface, you can press Ctrl-Alt-F7). If you are running Mac OSX, you can run the Terminal application (typically found under *Applications >Utilities*).

If you are running Windows, there are some command line options (typically called *cmd* or *command*, depending the version of Windows), however, these tend to be quite simplistic by UNIX standards. In fact, the Windows command prompt behaves entirely differently and uses different commands than a Unix shell. You could install a tool called Cygwin, which provides the basics of a UNIX environment if you wanted. However, for SCI, you will want to just use SSH (in MobaXTerm) to login to the MLP. We will explain how to get an use MobaXTerm soon.

Once you have started your terminal, it should display a *command prompt* (or just "prompt" for short). The prompt not only lets you know that the shell is ready for you to give it a command, but also provides some information. In this course, it does not display user or host information, but it would on your machine. Next is the current *directory*. In this case, the current directory is *~/sci/00_submit*. The ~ is UNIX shorthand for "your home directory" (which we will elaborate on momentarily). Then the learn2prog directory is inside of ~, and then the 00_submit directory is inside of that. After that, the $ is the typical symbol for the end of the prompt for a typical user, indicating that a command can be entered. The white box is the cursor, which indicates where you are typing input.

The prompt displays this information since it is typically useful to know immediately without having to run a command to find out. While it may seem trivial to remember who you are, or what computer you are on, it is quite common to work across multiple computers. For example, a developer may have one terminal open on their local computer, one logged into a server shared by their development team, and a third logged into a system for experimentation and testing. Likewise, one may have multiple usernames on the same system for different purposes. Exactly what information the prompt displays is configurable, which we will discuss briefly later.

Command Line Arguments

Many UNIX commands take arguments on the command line to specify exactly what they should do. In general, command line arguments are separated from the command name (and each other) by white space (one or more spaces or tabs). For example, *ls -a* .. will display "all" files in the parent directory, including those that are usually not displayed.

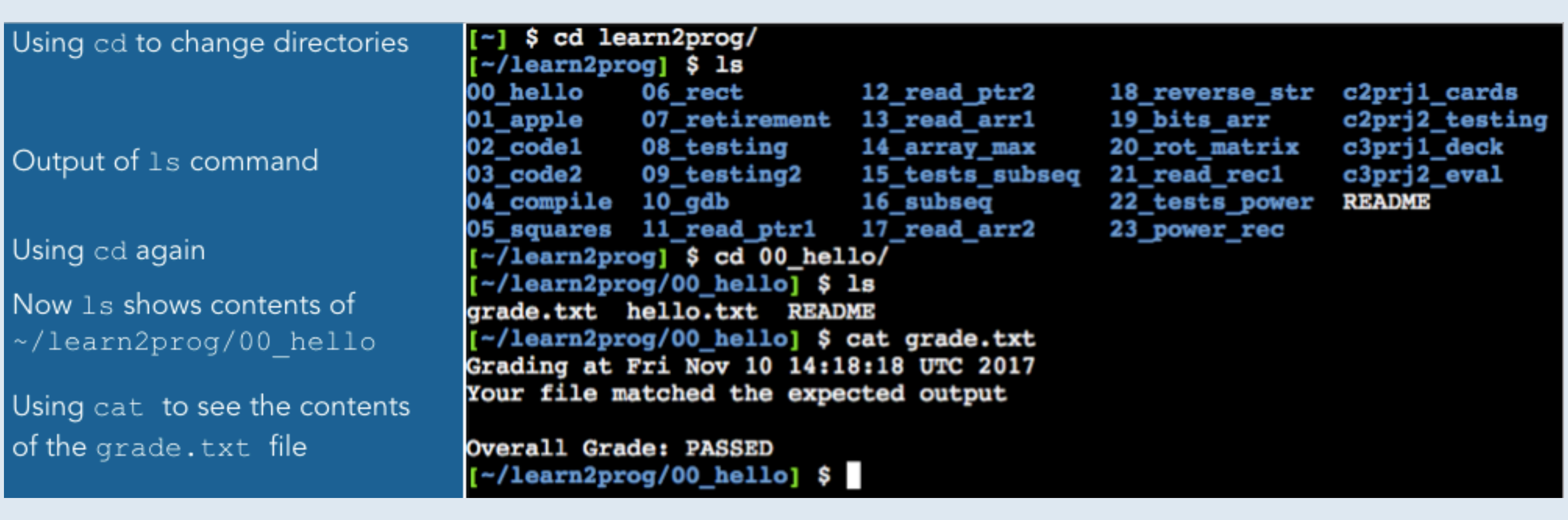
The ".." is an argument that tells *ls* which directory to display the contents of. The argument "*-a*" is an example of an "option." Options are arguments that differ from "normal" arguments in that they start with a hyphen "-" and change the behavior of the command, rather than specifying the typical details of the program.

Directories

The discussion of the prompt introduced three important concepts: *directories*, the *current directory*, and the user's *home directory*. Directories are an organizational unit on the *filesystem,* which contain files and/or other directories. You may be familiar with the concept under the name "folder", which is the graphical metaphor for the directory. The actual technical term, which is the correct way to refer to the organizational unit on the filesystem is "directory". Folder is really only appropriate when referring to the iconography used in many graphical interfaces.

To understand the importance of the "current directory," we must first understand the concept of *path names*—how we specify a particular file or directory. In UNIX, the filesystem is organized in a hierarchical structure, starting from the *root*, which is called / . Inside the root directory, there are other directories and files. The directories may themselves contain more directories and files, and so on. Each file (or directory—directories are actually a special type of file) can be named with a *path*. A path is how to locate the file in the system. An *absolute path name* specifies all of the directories that must be traversed, starting at the root. Components of a path name are separated by a / . For example, */home/drew/myfile.txt* is an absolute pathname, which specifies the *myfile.txt* inside of the *drew* directory, which is itself inside of the *home* directory, inside the root directory of the file system.

The "current directory" (also called the "current working directory" or "working directory") of a program is the directory which a *relative path name* starts from. A relative path name is a path name which does not begin with / (path names which begin with / are absolute path names). Effectively, a relative path name is turned into an absolute path name by prepending the path to the current directory to the front of it. That is, if the current working directory is /home/drew then the relative path name *textbook/chapter4.tex* refers to */home/drew/textbook/chapter4.tex.*



All programs have a current directory, including the command shell. When you first start your command shell, its current directory is your *home directory*. You can see this in the image above, where the prompt on the first line is [~].On a UNIX system, each user has a home directory, which is where they store their files. Typically the name of user's home directory matches their user name. On Linux systems, they are typically found in */home* (so a user named "drew" would have a home directory of */home/drew*). Mac OSX typically places the home directories in */Users* (so "drew" would have */Users/drew*). The home directory is important enough that it has its own abbreviation, ~. Using ~ by itself refers to your own home directory. Using ~ immediately followed by a user name refers to the home directory of that user (e.g., *~fred* would refer to fred's home directory).

There are a handful of useful directory-related commands that you should know. The first is *cd*, which stands for "change directory." This command changes the current directory to a different directory that you specify as its command line argument (recall from earlier that command line arguments are written on the command line after the command name and are separated from it by white space). For example, *cd /* would change the current directory to / (the root of the filesystem). Note that without the space (*cd/*) the command shell interprets it as a command named "cd/" with no arguments, and gives an error message that it cannot find the command. In the image above, you can see two uses of the cd command. The first changes the current directory from the home directory to the learn2prog directory. The second changes to the 00_hello directory (which is inside of the learn2prog directory).

The argument to cd can be the pathname (relative or absolute—as a general rule, you can use either) for any directory that you have permission to access. We will discuss permissions in more detail shortly, but for now, it will suffice to say that if you do not have permission to access the directory that you request, *cd* will give you an error message and not change the directory.

Another useful command is *ls* which lists the contents of a directory—what files and directories are inside of it. With no arguments, ls lists the contents of the current directory. If you specify one or more path names as arguments, *ls* will list information about them. For path names that specify directories, *ls* will display the contents of the directories. For path names that specify regular files, *ls* will list information about the files named. You can see two examples of *ls* in the image above. The first shows the contents of the learn2prog directory (here, a list of assignments). The second shows the contents of the 00_hello directory: a README with the instructions for the assignment, hello.txt which is the deliverable for this assignment, and grade.txt which gives feedback on your work).

The previous video showed an example of using the *cd* and *ls* commands. The first command in the example is *cd learn2prog*, which changes the current directory to the relative path examples. Then the prompt showed the current directory as *~/learn2prog*. The second command is *ls*, which lists the contents of the examples directory (since there are no arguments, ls lists the current directory's contents). In this example, the current directory has a directory (00_hello) and a regular file (README) in it. The default on most systems is for *ls* to color code its output: directories are shown in dark blue, while regular files are shown in plain white. There are other file types, which are also shown in different colors.

The *ls* command also can take special arguments called "options". For example, for *ls* the *-l* option requests that *ls* print extra information about each file that it lists. The *-a* option requests that ls list all files. By contrast, its default behavior is to skip over files whose names begin with . (i.e., a dot). While this behavior may seem odd, it arises from the UNIX convention that files are named with a . if and only if you typically do not want to see them. One common use of these "dot files" is for configuration files (or directories). For example, a command shell (which parses and executes the commands you type at the prompt) maintains a configuration file in each user's home directory. For the command shell *bash*, this file is called *.bashrc.* For the command shell *tsch*, this file is called *.cshrc.*

The other common files whose names start with . are the special directory names . and .. . In any directory, . refers to that directory itself (so *cd .* would do nothing—it would change to the directory you are already in). This name can be useful when you need to explicitly specify something in the current directory (*./myCommand*). The name .. refers to the *parent* directory of the current directory—that is, the directory that this directory is inside of. Using *cd ..* takes you "one level up" in the directory hierarchy. The exception to this is the .. in the root directory, which refers back to the root directory itself, since you cannot go "up" any higher.

The *ls* command has many other options, as do many UNIX commands. Over time, you will become familiar with the options that you use frequently. However, you may wonder how you find out about other options that you do not know about. Like most UNIX commands, *ls* has a man page (as we discussed previously) which describes how to use the command, as well as the various options it takes. You can read this manual page by typing *man ls* at the command prompt.

Two other useful directory-related commands are *mkdir* and *rmdir*. The mkdir command takes one argument and creates a directory by the specified name. The rmdir command takes one argument and removes (deletes) the specified directory. To delete a directory using rmdir, the directory must be empty (it must contain no files or directories, except for . and .. which cannot be deleted).

More UNIX Commands

Displaying Files

Now that we have the basics of directories, we will learn some useful commands to manipulate regular files. We will start with commands to display the contents of files: *cat, more, less, head, and tail.*

The first of these, *cat,* reads one or more files, concatenates them together (which is where it gets its name), and prints them out. As you may have guessed by now, *cat* determines which file(s) to read and print based on its command line arguments. It will print out each file you name, in the order that you name them.

If you do not give *cat* any command line arguments, then it will read *standard input* and print it out. Typically, standard input is the input of the terminal that you run a program from—meaning it is usually what you type. If you just run *cat* with no arguments, this means it will print back what you type in. While that may sound somewhat useless, it can become more useful when either standard input or standard output (where it prints: typically the terminal's screen) are *redirected* or *piped* somewhere else.

While you can use *cat* to display the contents of a file, you typically want a bit more functionality than just printing the file out. The *more* command displays one screenfull and then waits until you press a key before displaying the next screenfull. It gets its name from the fact that it prompts—*More*—to indicate that you should press a key to see more text. The *less* command supercedes *more* and provides more functionality: you can scroll up and down with the arrow keys, and search for text. Many systems actually run *less* whenever you ask for *more*.

There are also commands to show just the start *(head)* or just the end *(tail)* of a file. Each of these commands can take an argument of how many lines to display from the requested file. Of course, for full details on any of these commands, see their *man* pages.

Note that these commands just let you view the contents of files.

Moving, Copying, and Deleting

Another task you may wish to perform is to move *(mv)*, copy *(cp)*, or delete (*rm*—stands for "remove") files. The first two take a source and a destination, in that order. That is where to move (or copy) the file from, followed by where to move (or copy) it to. If you give either of these commands more than 2 arguments, they assume that the first N-1 are sources, and the last is the destination, which must be a directory. In this case, each of the sources is moved (or copied) into that directory, keeping its original filename.

The *rm* command takes any number of arguments, and deletes each file that you specify. If you want to delete a directory, you can use the *rmdir* command instead. If you use *rmdir*, the directory must be empty—it must contain no files or subdirectories (other than . and ..). You can also use *rm* to recursively delete all files and directories contained within a directory by giving it the *-r* option. Use *rm* with care: once you delete something, it is gone.

Pattern Expansion: Globbing and Braces

You may (frequently) find yourself wishing to manipulate many files at once that conform to some pattern—for example, removing all files whose name ends with ~ (editors typically make backup files while you edit by appending ~ to the name). You may have many of these files, and typing in all of their names would be tedious.

Because these names follow a pattern, you can use *globbing*—patterns which expand to multiple arguments based on the file names in the current directory—to describe them succinctly. In this particular case, you could do *rm* *~ (note there is no space between the * and the ~; doing *rm* * ~ would expand the * to all files in the directory, and then ~ would be a separate argument after all of them). Here, * is a pattern which means "match anything". The entire pattern *~ matches any file name (in the current directory) whose name ends with ~. The shell expands the glob before passing the command line arguments to *rm*—that is, it will replace *~ with the appropriately matching names, and *rm* will see all of those names as its command line arguments.

There are some other UNIX globbing patterns besides just *. One of them is ? which matches any one character. By contrast, * matches any number (including 0) of characters. You can also specify a certain set of characters to match with [...] or to exclude with [!...]. For example, if you were to use the pattern file0[123].txt it would match file01.txt, file02.txt, and file03.txt. If you did file0[!123].txt, then it would not match those names, but would match names like file09.txt, file0x.txt, or file0..txt (and many others).

Sometimes, you may wish to use one of these special characters literally—that is, you might want to use * to mean just the character *. In this case, you can escape the character to remove its special meaning. For example, rm * will remove exactly the file named *, whereas rm * will remove all files in the current directory.

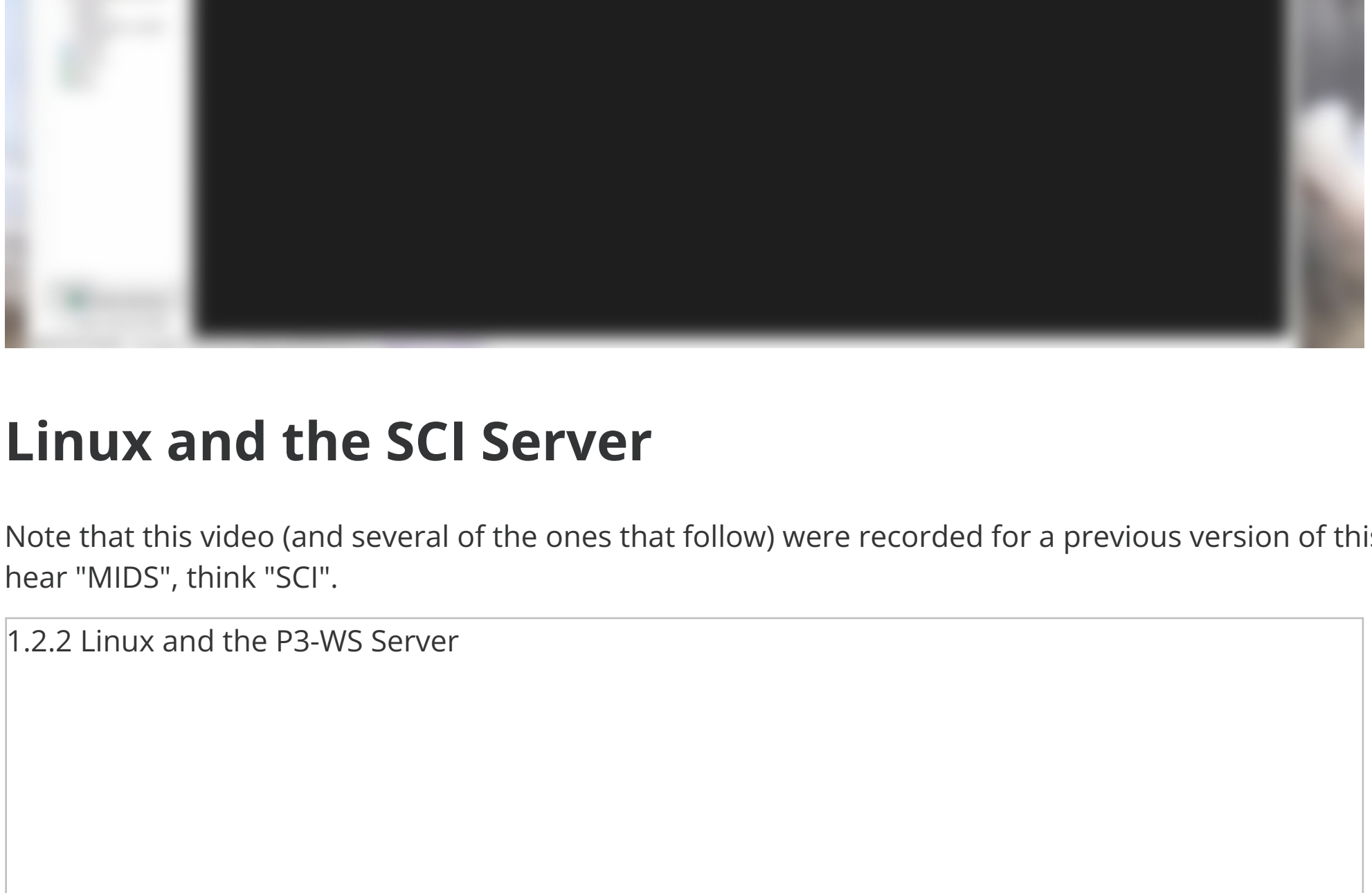
SCI Server

From now on, you will need a place to practice what you are learning, so we are providing a Linux server for this course, which you can log in to in order to do everything from practice the UNIX commands you just learned to access the programming assignments that are a part of this course.

The next two videos explain how to log in. If you are using a Mac or Linux, you can open a terminal and skip the next video. If you are running Windows, you will want to watch the next video to show you how to install an SSH client.

Go ahead and find the email with subject "SCI Assignment Log-In Information," which contains your username and password.

SSH for Windows Users



Linux and the SCI Server

Note that this video (and several of the ones that follow) were recorded for a previous version of this course that used a different server name. Whenever you see or hear "MIDS", think "SCI".

1.2.2 Linux and the P3-WS Server