



Lab 1

 Lab 1 assignment is due on Friday, February 21, by 9pm in your svn repository. For help, please check the [Labs FAQ](#) first. Specifically, please read the Terminal Troubleshooting and SVN Troubleshooting.

Please ask all questions about this assignment during the office hours.

 This lab is to be done in the **Linux computer lab** during **any** time of the week, or **on your own laptop** after you install the required [Virtual machine](#). Plan your time accordingly since late submissions will not be accepted. **No exceptions**. If you have any issues, contact the instructor during office hours.

Introduction to Unix

The purpose of this lab is to introduce you to the computing resources we will be using in ECE 120 this semester.

By the end of this lab, you will be able to:

- Navigate the Linux directory structure through the command line interface.
- Add, remove, and edit files using the command line terminal.
- Use Subversion to maintain a backup copy of your work.

There's a lot of reading to do for this week's lab but hang in there and make sure you understand all of the concepts. They will make the lab assignment (and the rest of the semester) much easier.


It is strongly advised that you install the [course virtual machine](#) on your own laptop and use it for the labs.

Linux interface

The computers in our classrooms are running **Linux**. Linux is a "**Unix-like**" **operating system**. An **operating system** is the most basic software running on a computer. It manages the computer's resources, controls peripherals, and executes **applications**. Other examples of operating systems are Windows and Mac OS. **Unix** was originally developed by Bell Labs in 1969 and has since split off into a variety of different operating systems which have been very significant in computing throughout the decades.

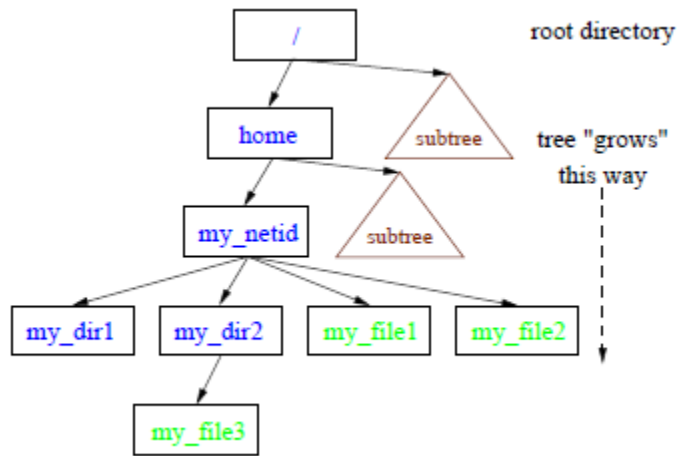
The Linux computers in your lab are managed by your Department. **When you are logged in to a Linux computer in the lab, it does not matter which particular computer you are sitting at; all of your personal files will remain the same.**

If you haven't already done so, log in to your computer using your netid and password.

 If you can't log in, let the lab instructor know.

Graphical User Interface

The desktop environment is based on the mouse, windows, and icons. You can double-click on an icon, like your home folder, and a window opens up showing the contents. Because of the graphical nature of the desktop environment, we call it a **graphical user interface** or **GUI** for short. Inside a folder, you will find some documents. You may also find more folders, which in turn have contents. This is the **filesystem hierarchy**. The filesystem is like a tree where each folder is a branch of the tree and the leaves are the documents. On a Unix system, folders are known as **directories**, and documents are known as **files**. Directories inside directories are called **subdirectories**.



The GUI also gives you the ability to create files and directories.

In your home directory, create a subdirectory called "part1". Open that directory and create a text file called "foo.txt". The GUI is designed to be intuitive, so we won't describe here how to do these things.

The GUI has limitations, however. Anything that the GUI programmers thought of is easy to do. If you try to do something the GUI programmers didn't think of, however, the process can be painful. For example, create a subdirectory called "years" inside "part1". For each year from 1900 to the present year, create a file in "years" named after that year with ".txt" appended to the end. For example, the first three files should be named "1900.txt" "1901.txt" "1902.txt".



Unless you like repetitive work, you don't have to create all of the files. Once you have seen how much time this would take, please stop and move on.

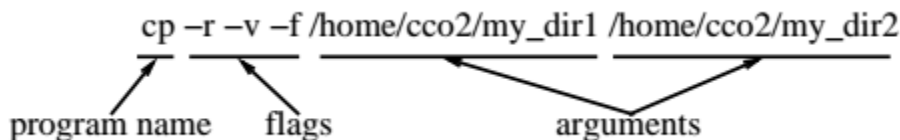
Command Line Interface

Most operating systems have an alternative interface, called a **command line interface** or **CLI** for short. In this interface, instead of telling the system what to do by clicking on icons, you give it text commands by typing them on the keyboard, and the computer responds in the same way with a textual response.

To open a command line terminal window, find "Applications" on the menu bar, and click on "System Tools" -> "Terminal". A window should pop up with a **prompt** that will look something like this:

```
[netid@linux3 ~]$
```

In a Unix command line, commands have a basic format which is *the name of the program* followed by a space, followed by a set of optional flags, followed by *the first argument* (if any), followed by a space, followed by any additional arguments, all separated by spaces (see the line of code below). An **argument** is also known as an option or a parameter.



More generally, a command will look like this:

```
program [arg1] ...
```

The above is called a *synopsis*; it describes command line syntax in an abstract way. It is not meant to be taken literally; each word is a placeholder for the name of real programs and arguments. [Words in square braces] represent optional arguments. Note that if you use these optional arguments, the square brackets should *not* be included. The "..." means that optional additional arguments can follow.

Now we will take a look at some useful Unix commands.

echo

A simple program to use is called **echo**. This program simply takes its arguments and echoes them to the terminal, hence its name. Try running the following (copy it exactly):

```
echo "Can you hear me?"
```

Instead of typing this manually, most terminal programs allow you to copy and past the text directly into the command prompt. Usually control-shift-v will do the trick. Don't forget to hit return after entering the command; that tells the terminal that you're done typing. Notice that we put quotes around the argument to the **echo** program. Running inside the terminal there is a program, called the **shell**, which reads and interprets the command that you type, breaking it up into programs and arguments. Quotes tell the shell to interpret "Can you hear me?" as one argument rather than 4 and to take the "?" character literally (the shell usually takes "?" to mean something special). If you don't have any spaces or special characters in your argument, you don't need quotes. (| & ; () < > as well as spaces and tabs are considered "special characters")

pwd

Another simple program is called **pwd**, short for "print working directory". The working directory is the command line's notion of where you are in the filesystem hierarchy. This is similar to the directory represented by the window you have open in the GUI. This program takes no arguments, so you just run it like so:

```
pwd
```

You should see output like the following:

```
/home/netid
```

This is called a **path**. It is a compact description of how to find the directory that you're currently in. Each step in the traversal is separated by a "/". It says to find this directory, start at the **root** of the filesystem (represented by the leading "/"), go into a subdirectory called "home" and then go into a subdirectory named for your netid. Any path which starts at the root of the filesystem (has a leading "/") is called an **absolute path** since it is anchored to the root (like the trunk of the tree; every branch comes from it). If you like, you can confirm this description is correct with the GUI.

Command history

With the terminal in focus, hit the up arrow a few times. Notice that the previous commands you entered appear. You can use the up/down arrow keys to navigate through your command **history**, and the left/right arrow keys to move your cursor and modify the command before resubmitting it for execution. This trick becomes useful when you want to run a series of similar commands, like in the following section.

ls

Suppose we want to know the contents of a directory. The **ls** program (short for "list directory contents") is designed to do exactly that. It can take zero or more arguments:

```
ls [directory] ...
```

If run with no arguments, it just lists the contents of the current working directory:

```
[netid@linux3 ~]$ ls  
part1 notes.txt
```

If run with arguments, it interprets the arguments as directories (folders) and lists the contents of those directories.

```
[netid@linux3 ~]$ ls part1  
foo.txt
```



Replace [directory] with the directory that you wish to list the contents of. That is, you should not type

```
ls directory
```

unless you have a folder called directory!

The directory argument is actually a **path**. With no leading "/" character it is a **relative path**, meaning that it is followed relative to the current working directory. We can also use absolute paths:

```
[netid@linux3 ~]$ ls /home/netid/part1
foo.txt
```

The code above will have the same output as running

```
[netid@linux3 ~]$ ls part1
foo.txt
```

if your current working directory is

```
[netid@linux3 ~]$ pwd
/home/netid
```

This is because the two paths actually point to the same folder.

In each directory, there are also some special directories to help you navigate the filesystem, but they are hidden by default (on the GUI as well as the CLI). To show them, we run **ls** with "-a" as an argument:

```
ls -a
```

Single character arguments preceded by a "-" character are called **flags**. They are used to modify the behavior of a program. In this case, the "-a" flag tells **ls** to show hidden files and directories. In Unix, these are any files or directories that start with a "." character. You will see output like the following:

```
.
..
.adobe
.bash_history
.bash_profile
.bashrc
part1
```

We're interested in the first two entries, the rest are hidden configuration files used by programs running on the system. The first directory is called "." It always circularly references the directory that it's in. This may seem strange, but it can be useful. For example, if **ls** didn't work without any arguments, we could still list the contents of the current working directory with:

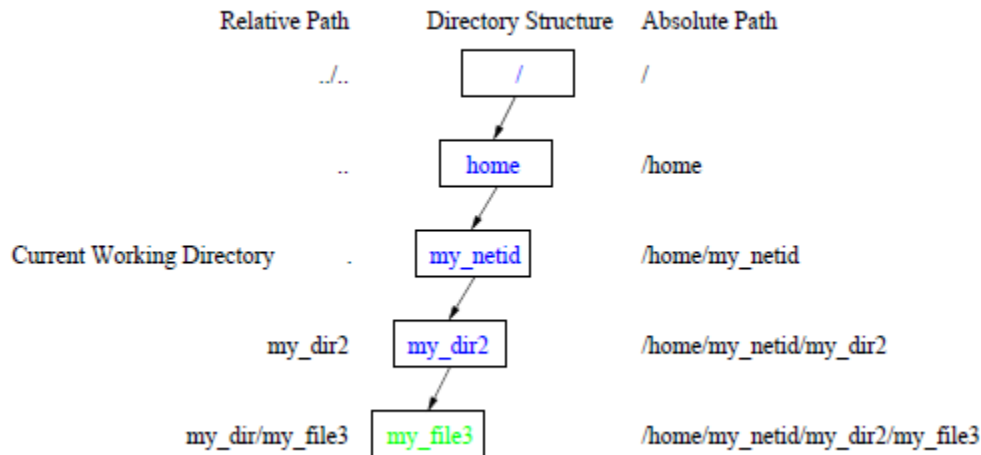
```
ls .
```

The second entry is "..", which is a reference to the **parent directory**. This is the directory which contains the directory that it's in. We use this entry to navigate down the filesystem tree toward the root. Let's see what happens when we pass it as an argument to **ls**:

```
[netid@linux3 ~]$ ls ..
aolinux2  barakat2  chae14    feeney3   huang166  kacampb2  mitros1   oldani1   tdeleon2  wang429
aplewis2  bmmoore   cheng88   gambill   huff10    kurkows1  mtrower2  park379   thansen3  wfagen2
```

This looks like a list of other people's home directories, which makes sense since we were in /home/netid so the parent directory is /home. Make sense?

Following is an image that further illustrate the directory structure.



Let's play with some fancier paths. Make sure you are in your home directory

```
cd ~
```

and try the following with ls:

```
ls /
ls ../../home
ls part1/..
ls ./part1/.
ls ../../home/../../home
ls ../part1/../../../../
```

Can you explain how each path is evaluated? Be sure to ask for help if you don't understand.



It's helpful to use **ls** often. It's always good to verify the contents of your current working directory.

cd

If we want to go to another directory, we use the **cd** program, short for "Change working directory". The syntax of **cd** is simply:

```
cd directory
```

For example, to go into the **part1** directory (from your home directory) and list the contents, we can do:

```
cd part1
ls
```

Now what happens when you run "ls .." ?
How do you go back to the parent directory?

Tab completion

Another useful command line trick is tab completion. Whenever you're typing a **path** into the terminal, you can just enter the first few characters, hit the tab key, and the shell will attempt to complete the rest for you. You can use this trick as often as you like in a command. On some better shells, hitting tab twice will also show you a list of possible completions.

Try going to **part1** and entering "cd y" and then hitting tab to see tab completion in action. The shell should fill in "years" – the directory you made at the beginning of the lab using the GUI.

mkdir

This program creates new directories (short for "make directory").

```
mkdir directory ...
```

Use **mkdir** to create a new directory inside "part1" called "test". "mkdir" is the name of the function and "directory" is the first argument. The "directory" argument here will be the **path** of the directory that you want to make. So if you are inside of the "part1" directory, typing

```
mkdir test
```

will create a folder in "part1" called "test." You can always use absolute paths as well such as

```
mkdir /home/netid/part1/test
```

(if your netid is netid!).

Note that on the command line, no output is often a sign of success. If there was an error, the program will usually print out an error message giving a hint as to the problem. Use **ls** or the GUI to verify that you succeeded.

touch

This program creates a new empty file (or "touches" existing files to update their modification time).

```
touch file1 ...
```

Use **touch** to create a new file inside "part1" called "notes.txt." Again, the file1 argument is the **path** to a file. If you are inside of the "part1" directory, you can use

```
touch notes.txt
```

mv

This program **moves** files and directories to new locations.

```
mv source destination
```

Both of the arguments will be **paths**. **source** can be a file or directory. **destination** can be an existing directory, or a path to a new file that doesn't exist yet. If the destination is a directory, then the source file or directory will be moved into it. If the destination doesn't yet exist, then the source will be moved and renamed to the destination.



If the destination already exists, it will be overwritten! Be careful when using mv

Use **mv** to move "notes.txt" into "test". "notes.txt" was the file that we just made with "touch." "test" was the directory that we made previously with "mkdir." So to move "notes.txt" into the "test" directory if you are in the "part1" directory, type

```
mv notes.txt test
```

cp

This program **copies** files and directories.

```
cp [-r] source destination
```

The "cp" program behaves the same as **mv**, except that it duplicates files and directories rather than moving them. *The optional "-r" flag is needed when you want to copy directories and their contents.*



If the destination already exists, it will be overwritten!



For commands like `cp [-r]` that include the `[-r]`, do *not* include the brackets! That means that you will type `cp -r`

Use `cp` to make a copy of "test" called "test_copy".

rm

This program **removes** files and directories.

```
rm [-r] file1 ...
```



Be very careful when using the "rm" program. Once a file is gone, it's gone; there is no recycle bin to find your removed files.

As with the `cp` program, the "-r" flag is used to a remove a directory and its contents.

Use "rm" to remove "test_copy" and its contents.

Getting help

To get very detailed syntax for a particular program, use the **man** utility:

```
man [section] program
```

For example, to get help on using "rm", run:

```
man 1 rm
```

To scroll down, hit the return key. *To exit, hit "q".* On better configured systems, you can also scroll with the arrow keys.

Note that we're using section 1. This is the section for programs used on the CLI; all CLI programs (including the ones we've introduced in this lab) can be found in section 1. When we get into C programming, we'll be looking at other sections for C functions, but for now, everything can be found in section 1.

Text editing

For the purposes of ECE 120 work, we suggest using **gedit** for editing text files. **gedit** is a simple, GUI based text editor that you can also invoke from the command line like so:

```
gedit [file]
```

Try running `gedit` and use it to put some notes into "notes.txt". Then save your work and close the window. Notice that the command line doesn't become available until after the window is closed. You can change this behaviour by adding the special character "&" to the end of the command like so:

```
[netid@linux3 test]$ gedit notes.txt &
```

Something like the following will appear on the command line as `gedit` opens.

```
[1] 14978
```

The special character "&" tells the shell to create a separate **process** (task) for the program being run so that shell can accept commands at the same time the program is running. When the shell launches a separate process, it also prints the **job number** (1 in the example) and **process ID** (14978 in this example). We can use these identifiers to control the process; more on that later.

If we just want to quickly see what's in a file quickly without opening it, we can use the `cat` program (short for concatenate and print):

```
cat file1 ...
```

Use **cat** to see the contents of "notes.txt"

Many alternative CLI and GUI based text editors are available. Use the editor you're most comfortable with (except for some Windows editors, see below). Here's a short list:

nano (Unix)

A CLI based text editor. **nano** is relatively easy to use, but has limited features. This is the editor we recommend students to use if a graphical text editor like **gedit** is unavailable.

vi or vim (Unix)

A CLI based text editor available almost every Unix system. Rather difficult to use. Has a GUI version called "gvim".

emacs (Unix)

A CLI based text editor that has every feature imaginable. Also has a GUI version called "xemacs".

Other operating systems

Mac OS X

Mac OS X is Unix based, so you can run a terminal and use all the above editors. In addition, Apple's Xcode program is a good text editor.

Windows

We recommend Notepad++ for work on Windows. **Do not** use MS word, Notepad or Wordpad. These text editors will produce certain characters that can break programs used in this class. For more information about this, see the [Coding Conventions](#).

Revision Control (Subversion)

Revision control (or version control) is the management of changes to files. Have you ever made a change to a document that you later regretted? A revision control system allows you to keep track of the different versions of a document, so that you can later revert back to before you made that unwanted change. Revision control is especially useful in team projects where multiple people are making changes to a set of documents due to its ability to automatically handle separate changes to the same file.

In this class, we will be using a revision control system called Subversion. Subversion maintains a central **repository**, one or more revisions of a directory and all of its contents, on a server. You can get a copy of one version of that directory in a process called a **checkout**. You can make **local changes** to your copy of the directory, and then add the new version to the central repository in a process called a **commit**.

Checkout

The program for doing all work in subversion is called "svn". We've already set up a central repository for you. You can view the contents of your repository using a web browser; just go to:

```
https://svn.intl.zju.edu.cn/#!/#SPR20-ECE120/view/head/netid
```

where **netid** is your INTL NetID, typically your first name followed by .17, e.g., Bo.17. After being prompted for your password, you should see a directory called "lab1"

Let's "checkout" a copy of your repository. The checkout syntax is as follows:

```
svn checkout url directory
```

From *your home directory*, you'll want to do something like this:

To get to your home directory, type:

```
cd ~
```

To checkout a copy of your Subversion repository, type:


```
svn checkout https://svn.intl.zju.edu.cn/svn/SPR20-ECE120/netid ece120sp20 --username netid
```


Don't forget to change **netid** to your actual ID. For example:


```
cd ~
svn checkout https://svn.intl.zju.edu.cn/svn/SPR20-ECE120/KaiwenC.17 ece120sp20 --username KaiwenC.17
```

This would checkout the course files and place them in a folder called ece120sp20 inside of your home directory.

This process just created a **working copy** (a copy for you to "work on") of your repository, on your computer. The **working copy** is now in a directory called "ece120sp20".

 You will be asked to type in your password. *No characters will show up on the screen while you are typing your password.* Type in your password as normal and press enter. Additionally, make sure you type it right the first time, pressing backspace will not let you correct a typo in your password.

 You might get a message warning about storing plaintext passwords. Since it's generally not a good idea to leave your password lying around unencrypted, we recommend you deny plaintext storage.

 A number of other warnings and errors may come up while you are learning SVN. Before you post a question regarding these error messages, check [SVN Troubleshooting](#), or the questions already asked and answered by other students below, the answer to your issue may be already there.

Lab 1 assignment

The purpose of this lab assignment is to give you some more practice with the concepts explained above. Reference the introductory text if you get stuck. If you can't figure it out, please go to office hours to ask a TA or UA.

You should now have a directory called "ece120". Inside should be the directory "lab1" which you saw earlier from your web browser.

Go into this directory.

Modify

Open foo.txt with your text editor, remove the last line, and save this change.

Run "svn status" to see what has changed in the your local **working copy** of the repository

```
[netid@linux3 lab1] svn status
M      foo.txt
```

The "M" means that the file has been modified.

Commit

```
svn commit -m "Removed last line of foo.txt"
```

In this case, the "-m" (for message) is now a flag with an argument. It specifies a message to be associated with the new version you are sending to the server. These messages will be useful if you ever realize you made a mistake and need to go back and find a particular revision, so make them good summaries of the changes you've made. Note that the "-m" flag is not optional. You must include a message every time you commit or SVN will complain.



If at this point, you're still getting annoying messages from svn about plaintext passwords, notice that the message tells you to modify a config file to make it go away. Edit that config file, and add:

```
store-plaintext-passwords = no
```

to the end of it.

Add

Create a file called "bar.txt" and put the text "This is a file" inside. Now run "svn status". Notice that there's a "?" next to the new file. This means subversion isn't sure if you want that file added to the repository. So we can't commit the file just yet. First we need to run:

```
svn add bar.txt
```

Now when we run "svn status" we see an "A" for added next to "bar.txt". Now we can commit the new file; go ahead and do it.

Move, remove, and copy

When we're working with a *local copy* of a subversion repository, the best way to do file moving operations is to use the subversion *wrapped* version of these programs, so that subversion knows what's going on and doesn't get confused. A *wrapped* version of a program is one that does what the original does with some additional functionality. These commands will work just like the originals, except that subversion does some additional bookkeeping to track what has changed:

```
svn mv source destination
svn cp source destination
svn rm file1 ...
```

Use the subversion commands to move "baz.txt" inside "stuff". Copy "junk.txt" to "stuff" as well. Finally, remove "garbage.txt". Run "svn status" to see the changes you're about to commit. A "D" indicates a deleted file (subversion thinks of a move as a deletion followed by an addition).

If the changes look good, go ahead and commit.

Revert

If you make a mistake **that you haven't committed**, you can revert your changes back to the last commit with the revert command:

```
svn revert [-R] file_or_directory ...
```

You can revert individual files, or entire directories. You'll need to use the "-R" flag to revert directories and their contents. Notice that this is a capital R instead of a lowercase r. Unfortunately, different programs don't necessarily agree on the meaning of flags; you'll just have to remember that svn uses a capital R.

Practice reverting by making a change to a file and then reverting it; you should see the changes disappear (you may have to reload your text editor)

Update

When someone else commits changes to your repository, your **working copy** will become **out of date**. In this class, the teaching staff will add directories to your repository for future labs and machine problems. You will then need to **update** your working copy so that it reflects those changes.

To update your working copy of your repository, run the following command (from your "ece120" directory):

```
svn update
```

If you try running this now, nothing will happen; we haven't added any new content since you last checked out your repository, so your working copy is already up-to-date.

In next week's Lab this will be your first step so that you can get the files for Lab 2.

Checking your results

Look at the "lab1" directory. The hierarchy should look like the following:

- lab1
 - bar.txt
 - foo.txt
 - junk.txt
 - stuff
 - baz.txt
 - junk.txt

If it does, and the last line of "foo.txt" has been removed and "bar.txt" contains "This is a file", then you've successfully modified the files in your working copy. To verify that your changes have been successfully submitted to the repository, run 'svn status' in your lab1 folder. If nothing is printed, your working copy is current. Otherwise, 'svn status' will indicate which files have been modified locally and have not been submitted to the repository. Try to commit them again.



Get help from your instructor or another student if you're having trouble committing to the repository! Many of the labs in this course must be submitted by committing them to the repository. **We will not be accepting assignments sent via email.**

This constitutes your submission of Lab 1. Congratulations!

Advanced tricks

Remember that tedious task we had you do with the GUI? We're going to show you how to do it with the CLI.

Go to the "part1/years" directory with the CLI (to make it more spectacular, open up the same directory with the GUI) and copy and paste the following into the CLI (don't worry about understanding it):

```
seq -f "%.f.txt" 1900 `date +%Y` | xargs touch
```

If everything went well, you should see all the files appear in an instant (or what would be an instant, in a better filesystem). This command uses a number of programs and special shell characters to accomplish its goal. While it's certainly not intuitive, it is quite compact, and it gets the job done very quickly. This is the advantage and disadvantage of the CLI: it has a sharp learning curve, but once you become proficient at using it, it can be a very powerful tool.

Those interested in becoming proficient at using the command line can follow this nice tutorial: <http://linuxcommand.org/>

Useful references

Now that you have successfully completed this exercise, we expect you to use Linux command line interface and svn repository for this class as well as ECE 220 follow-up class. Go to [Resources](#) for a few references you may find useful to help you using Linux.