CMU MSP 36602 Intro to Linux on the Ubuntu Virtual Machine

H. Seltman, Feb. 6+11, 2019

1) Overview

a) **UNIX** is an **operating system** which is a competitor to and partial ancestor of MS-DOS, MS-Windows, and Mac OSX. Linux is a popular "flavor" or version of UNIX. There are versions of UNIX that run on almost any hardware platform.

b) A computer's operating system is the ***program that run all other programs*** on a computer and which manage resources. Typical **functions** include ***managing*** memory, file systems, the keyboard and mouse, other hardware, jobs, threads, inter-process communications (pipes), time slicing, interrupts, and user access. These core functions are part of the UNIX **kernel**.

c) UNIX also has a **command language** and a **scripting language,** which provide higher user-level functions than the kernel. This is via a **command shell** and **utility programs**.

2) Carl had created a **Linux virtual machine** that you can download as an appliance for VirtualBox (also used for SAS University Edition). See location and important instructions on Canvas. This gives you an Ubuntu Linux machine on your PC or Mac. (It also has a full Hadoop and Spark setup.)

3) **One time setup**

a) Start Virtual Box and choose "File / Preferences" on the menu, then under Input, uncheck "auto capture keyboard", and click OK.

b) Use File / Import appliance, then point to Hadoop2.ova, and import with default settings

c) Start Hadoop2 by clicking it and then clicking the Start arrow. What happens next varies a bit depending on whether you are doing a first time startup or restarting from a "shut down" state vs. restarting from a "save the machine state" state.

Wait many seconds until you get to the Ubuntu desktop, which provides some Linux Services without using the command prompt. While you are waiting, you can click any little "x" boxes on the upper right corresponding to messages about "mouse integration", etc.

d) It is recommended that you use full screen mode, so that it appears that your whole desktop is the Linux machine. To do this, choose "View / Full screen mode" from the menu. From the information box ***write down*** the information from the sentence that tells you what your VirtualBox ***Host Key*** is. You can click the box to prevent seeing this box in the future.

The box suggests that In the future, to ***exit full screen mode***, hold down the host key and press "F". To ***see the main menu bar***, hold down the host key and press the home key. It appear that this is only true if you do not uncheck "auto capture keyboard" as suggested in step 2a. If you unchecked the box as recommended, get to the main menu by ***holding your mouse at the bottom of the screen***, and select "View / Full Screen" to exit full screen.

e) On the menu, go to "Devices / Shared Clipboard", and select "Bidirectional" so that you can ***copy and paste*** text to and from the virtual machine and your computer. In Linux, highlight the text and right-click to get to "copy". Use shift-insert or control-y as the paste key.

f) **Setup a "share" folder to facilitate sharing files across "machines"**

i) Make a folder on your PC or Mac that will be used to share files across machines. Let's call in "share" (in some directory of your choice).

ii) On the VirtualBox menu, choose Machine/Settings. Choose "Shared folders" on the left, then click the folder "+" icon ("Adds new shared folder."). Under "Folder Path" navigate to the folder you want to share on your computer. Choose a "Folder Name", e.g., "share", and check "Auto-mount" and "Make Permanent", but not "Read-only". Click OK twice.

iii) Start a **terminal window** by clicking the icon that shows a terminal with ">_". This is where you will do most of your work. You can move and resize your terminal. You can start multiple terminals.

iv) Create a share folder on your VirtualBox by typing `mkdir /home/student/share`.

v) Connect the two folders by typing:

```
sudo mount -t vboxsf share /home/student/share
```

(If you called your "Folder Name" pointer to your Mac/Windows machine folder something other than "share", use that instead.) You will need to enter your VB password.

vi) Now when you put any file put in either /home/student/share (a.k.a. ~/share) on the VirtualBox or the "Folder Path" on your PC/Mac will also appear in the other location.

4) Working with the Ubuntu Linux Virtual Machine

a) Assuming that you have closed the your Ubuntu machine, on any given day, start by running Oracle VM Virtual Box and then double clicking on Hadoop2.

b) If you don't use the machine for a little while you will see a window with the header "Student" and the word "***password***". Enter your password in the password box.

c) The **Ubuntu desktop** is a bare bones GUI for Linux that lets you accomplish some things without using the command prompt. You should mostly use the command prompt to get more Linux practice. The icon bar on the left has the GUI functionality.

d) E.g., to start a ***Firefox browser***, so that you can look things up while working on your virtual Linux machine, click the Firefox world icon. If it is full screen, move your mouse to the top left and click the small rectangle inside the circle (next to the "x" and "-" circles) to make it a window. Resize and move the window using the mouse in the usual way.

e) To start a **terminal window** by click the icon that shows a terminal with ">_". This is where you will do most of your work. You can move and resize your terminal. You can start multiple terminals.

f) To **quit working with the Linux virtual machine**, use File / Close on the menu or click the corner "X". Normally you will chose "Save the machine state" to allow a much quicker start in the future.

5) Understanding the Linux file system

a) The file system is a ***single rooted tree*** (visualized with the root at the top). This contrasts with DOS/Windows where each drive has a drive letter and a tree (e.g., c:\). The root is designated "/". Disk drives may be **mounted** on the root as new branches off the root.

b) In a very real sense, ***everything on a UNIX system*** except a "process" ***is a file*** somewhere in the tree.

c) In the file system, a file is represented by a **name** and **inode** pair. The inode is a data structure that holds the physical locations of the file as well as properties such as owner; file type (see next item); permissions; date and time of creation, last read, and last change; file size; and more.

d) ***There are many kinds of files***:

i) **Regular files** hold **programs** in machine language or some intermediate "byte code"; or **data** in ASCII, utf-8, some other standard encoding, or in some other open standard format (e.g., jpg, IEEE floating point), or in some proprietary format. They may be human readable text or they may be "binary".

ii) **Directories** are special files that list other files.

iii) **Device files** are used for input and output, and come as "block devices" that have random access or "character devices" that handle data serially.

iv) **Links** are ways to make a file or directory visible in another part of the file system.

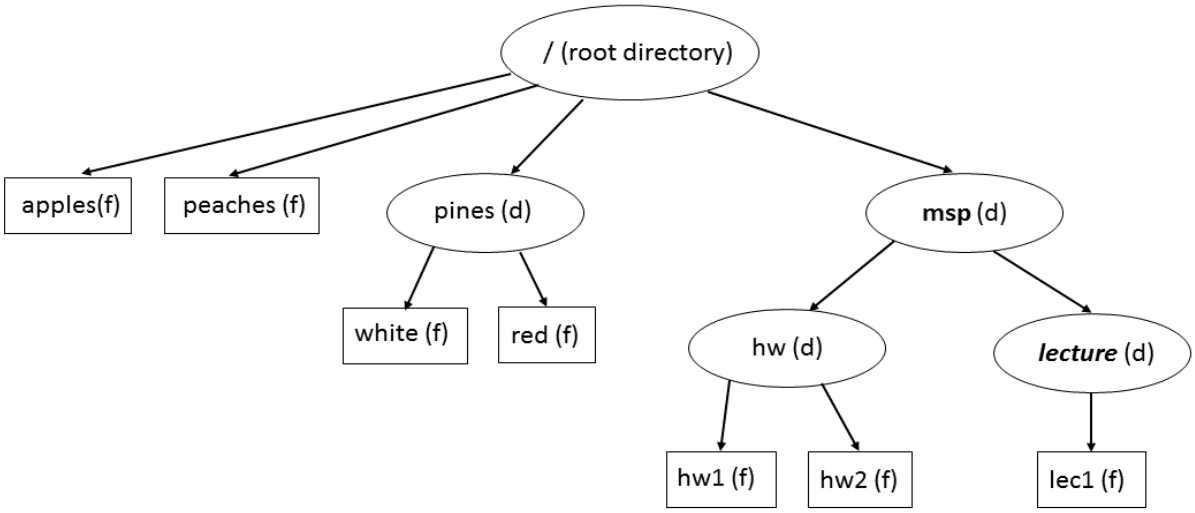
v) **Sockets** provide inter-process communication.

vi) **Named pipes** are an alternative to sockets.

e) Each user has a **home directory** in which they have full file rights. The `cd` command is ***used to change directories***. The `pwd` command shows the current **working directory**. The `ls` command is used to list the files in a directory.

f) ✶The directory structure may be traversed starting at the ***current directory***, the ***home directory***, or the ***root***. Note that traversing the directory does not change your current directory unless you use the `cd` command. The root directory is denoted by "/". The home directory is "~". A "." means the current directory. If a file reference starts with neither of these, it is interpreted as relative to the current (aka working) directory. In addition, a "/" is used to indicate a change in level of the tree (usually a child), and ".." means the parent directory ("up").

As an example, we have the following structure, where (d) indicates directory and (f) indicates a regular file, and "**msp**" is your home directory and "***lecture***" is your current working directory:



i) `/msp/hw/hw1`, `~/hw/hw1`, `../hw/hw1` all refer to the same file

ii) `/apples`, `~/../apples`, and `../../apples` all refer to the same file

iii) `lec1`, `./lec1`, `~/lecture/lec1`, `/msp/lecture/lec1`, and `/pines/../msp/hw/../lecture/lec1` all refer to the same file

g) The command `ls` is used to ***list the files*** in the current (working) directory or in any other directory for which you have access. Filenames starting with a period are normally not shown (they are "hidden"). Here are some examples:

i) `ls` or `ls .` lists files in the current directory

ii) `ls here/there` lists files in the "there" subdirectory of the "here" subdirectory of the current directory

iii) ls `../../foo/bar` lists goes two levels up the tree, then down two levels (to "foo" and then "bar"), and lists the files inside of "bar"

iv) `ls /home/stuff` lists the files in "stuff", two levels down from the root

v) `ls ~/two` lists files in the "two" subdirectory of the home directory

h) "?" and "*" are "one" and "any" wildcards. `ls a?b*x` matches "a-b12x", "a0bx", and "aabcccx", but not "abx", "macbx", or "a-b---xy". (Not this is **not** regex.)

6) **Options** for UNIX commands come between the command name and the command argument(s) and begin with a dash ("-").

   The command `ls –i` shows the inode numbers of each file in the current directory, and, e.g., `ls –i mySub` shows all inode numbers for each file in the "mySub" subdirectory of the current directory.

   The command `ls -l` (long) ***shows the file type*** in the first column as "-" for regular files, "d" directory, "l" for link, "s" for socket, "c" for character device, "b" for block device, and "p" for pipe. In addition, the long form shows permissions (see below), owner, size and date. This form can also work with or without a directory specification, e.g., `ls -l /bin` shows all of the files in the bin subdirectory of the root in the long form.

   The command `ls -a` shows "all" files, including the hidden ones.

   The command `ls -R` recursively shows files and directories.

   The command `ls -d */` shows all of the directories in the current directory.

7) Directory and file exploration in the terminal window

   a) Try `pwd` which "prints" your working directory

   b) Note that Linux commands may have several parts separated by spaces. E.g., `echo` prints text to your terminal when you add the text after "echo ". Try it.

   c) The full details of the available commands and their usage depend somewhat on which command shell you are running. All command shells have **shell variables**, which hold named text. To access the contents of the variable, put a dollar sign ($) in front of the variable name.

   d) Try `echo $SHELL` or `echo My shell is $SHELL` to find which command shell you are running.

   e) Try `echo $HOME` to find out what your home directory is.

   f) Try `ls` to see the names of the files in your working directory.

   g) List all of the files, including those files classified as "hidden" because they start with a period.

   h) List the files in the "conf" folder.

   i) Change into the "conf" folder so that it becomes your working directory.

   j) List the files in your new working directory.

   k) Without changing your working directory, list the files in your home directory three different ways.

   l) Go back to your home directory using either `cd ..` or `cd ~` or `cd /home/student`. How do each of these work?

   m) List the files in the root directory. List the files in one of the root's subfolders other than "home".

8) UNIX systems usually follow a **standard file system structure**. The following directories are some of the ones usually found in the root:

   a) **/bin** contains binary files, i.e., compiled programs
   b) **/dev** contains file representations of peripheral devices
   c) **/etc** contains system wide configuration files
   d) **/home** contains home directories for each user
   e) **/lib** contains libraries, i.e. compiled code shared across programs
   f) **/tmp** is a place for temporary files
   g) **/usr** once used to hold user home directories, but now holds programs that are commonly used, but not system critical
   h) **/usr/include** holds header files (text files containing information needed for C programming)
   i) **/usr/lib** holds libraries for general use
   j) **/usr/local** tends to hold programs not distributed with the operating system
   k) **/var** stand for variable and holds files than change often
   l) **/var/log** hold logs of system activity
   m) **/var/mail** holds incoming mail

9) You can ***create and delete directories*** under your home directory using `mkdir` and `rmdir` followed by the directory name. E.g., `mkdir myChild` makes an empty directory called child in the current directory, and mkdir `~/one/two/three` creates "three" if the home directory already contains a directory called "two" under a directory called "one". Removing directories only works if the directories are empty. (A recursive version of `rm` can remove files and directories, but is dangerous because there is no undo.)

   Try the following exercises:

   a) List all subdirectories of the current directory.

   b) Make a subdirectory called "aaa"

   c) List all subdirectories of the current directory.

   d) Remove "aaa"

   e) List all subdirectories of the current directory.

10) Each valid user for a system has a **username** (try `echo $USER` or `whoami`). Users may also belong to one or more groups of users. Every file has three sets of **file permissions,** one for each "class". The classes are "user", "group", and "other". For each class the read, write, and execute permissions may be set or unset (denied). The current permissions are visible using the long version of the list command. E.g., `ls -l` might show:

```
-rw-r--r--  1 hseltman users 17440 Jan 30 09:20 602Syllabus2017.docx
-rw-r--r--  1 hseltman users 80627 Jan 30 09:20 602Syllabus2017.pdf
drwxr-xr-x  2 hseltman users  4096 Feb 24 16:48 data
drwxr-xr-x  2 hseltman users  4096 Feb 24 16:47 fiscalMacros
```

The first two lines are for regular files, indicated by the initial "-". The next two lines are for directories. The nine characters after the file type are thought of as three sets of three characters each, with one set for each of "user", "group", and "other". Within each set "r" means that read access is granted, "w" means write access, "x" means execution access, and "-" means" no access.

So for the two regular files, the user can read and write (also delete) the files, but not execute them as programs, and the group members and all others may read the files but not write (or delete) or execute them.

For the two directories, the user can read the directory (list its files), and write the directory (add and delete files and remove the directory if it is empty), as well as move into the directory (execute permission). Nobody else cannot write to the directory.

11) The **chmod** command is used to ***change permissions***. The easiest form uses syntax like `chmod csa fileName`, where "c" is class (one or more of "u" for user, "g" for group, "o" for other, or "a" for all three), "s" is sign ("+" for adding the permission, "-" for removing it), and "a" is action (one or more of "r" for read, "w" for write, and "x" for execute. E.g., to prevent yourself from accidentally deleting a file, use `chmod u-w myFile`. To allow yourself and members of your group to execute a script file, use `chmod ug+x myScript`.

12) A **UNIX shell** is a command-line interpreter that provides a user interface based on typing commands. Users can choose which shell they will see initially by setting a line in their ".login" configuration file. The most common shells are sh (Bourne shell), and its three extensions: csh (C shell), ksh (Korn shell), and bash (Bourne Again shell). The differences between them are not very great, at least for beginners. Here we focus on **bash**, which has the following features. (A good manual is at http://mywiki.wooledge.org/BashGuide):

a) bash "code" can be ***entered at the command prompt or run from a file (script)***

b) "#" starts a **comment**

c) Everything in the shell is **case sensitive**.

d) **Shell variables** are strings that persist for an entire logon session. They can be viewed with `echo $myVar` (unless inside single quotes). Many special pre-defined environmental variables exist, such as SHELL, HOME, and PATH.

The code below shows all currently defined variables, sets variable "me" to "hjs", shows the value of "me", and prints the user's home directory. No spaces are allowed around the "="! Use `unset me` to erase a shell variable.

```
set
me=hjs
echo $me  # or ${me} if needed
echo $HOME is the value of '$HOME'
```

With the let command, arithmetic can be done:

```
let x=5*2+4  # no spaces
let y=x+1
echo $y
```

Note that the math is integer math (11/4=2).

e) You can **run compiled programs** by typing their names. More exactly, the "execute flag" on the file must be set, and, unless you specify the full relative or absolute PATH name, the program must be in the "PATH".

i) If the program name is followed by "&" the program runs "asynchronously" in a new shell and you will immediately get another command prompt in the original shell to continue working. Without the "&", the program runs "synchronously" in the current shell, and you will not get another command prompt until the program completes.

ii) The **PATH** can be viewed using `echo $PATH`. Here is an example:

/TOIL-U3/hseltman/bin:/bin:/usr/bin:/usr/statlocal/bin:/usr/X11R6/bin:/usr/bin/X11:/usr/local/bin:/usr/contributed/bin:.

iii) The PATH is a colon-separated list of directories. ***Programs specified without a PATH will be searched for through the PATH list***, starting at the first one (e.g., /TOIL-U3/hseltman/bin, above) until a program of that name is found. The first one found is run.

iv) You can use `which programName` to get the first PATH location containing the program.

v) Note that many UNIX programs require "x-windows" (aka X11) to run in a separate window rather than in the terminal window. This is all setup for the virtual machine. When accessing Linux without the VM, you may need to install x-windows on your computer (XQuartz for a Mac, XWin32 for a Tectia Shell on a PC). You may also need to set an environmental variable called "DISPLAY" for this to work.

f) The Linux command cat is used in the form `cat myFile` to print the contents of a file to your terminal. Try print the file "legal" found in the "etc" subfolder of the root folder. The `head` and `tail` programs work similarly.

g) Example: compile and run the "upper" program

i) Download http://www.stat.cmu.edu/~hseltman/602/code/upper.c with

wget http://www.stat.cmu.edu/~hseltman/602/code/upper.c

Briefly examine the C program code in upper.c using `cat.`

ii) Compile (and link) the program using the command (must be a regular dash):

```
gcc upper.c –o upper
```

This produces the file "upper" which contains the machine language (for the machine it was compiled on) for the program.

iii) Check that the program is executable with `ls -l upper`.

iv) Type `which upper` to find out if "upper" is or is not in the PATH defined for your computer. If it is shown, then just typing `upper` will run the program. If you get the "Command not found" message, then you must specify the PATH to run the program. This is most easily done by typing `./upper`.

v) The program waits for you to type some text, and then it outputs the upper case version of your text. This happens repeatedly until you enter an empty string.

h) You can ***run shell scripts and scripts for other interpreted languages*** that allow source code in a file by using the **shebang notation**. To do this create a plain text file. On the first line put shebang "#!" followed by the PATH name to the program, e.g., "#!/bin/bash" for a bash script on our virtual machine. Place commands for that program on the remaining lines of the file. Then make the program executable using a Linux command like `chmod u+x myScript`. If the current directory is in your PATH, just type the name of the script to run it. Otherwise use `./myScript` to run the script named "myScript" in the current directory. You can also use `#!/usr/bin/env myLanguage` to arrange for the "env" program (normally found in the "/usr/bin" directory) to locate and run "myLanguage".

   E.g., download "pyTest3" from the same location as upper.c (above). Set the permissions to allow execution, and then run the program. The commands in the file are run by Python and the output goes to your terminal.

i) The shell maintains a **history** of previously entered commands. Use `history` to see a list of previous commands with a command number and the time run also shown. Use, e.g., `!9` to rerun command number 9.

j) The shell has **tab completion**, so typing `his` then pressing the Tab key causes the shell to add `tory` to make `history`. This also works for file names. If there is more than one possible completion, Tab will not do anything, but a second Tab will show the different choices.

k) The ***shell can be customized***: any command placed in a file called "~/.bashrc" will be automatically run when the shell starts.

l) UNIX has a concept of **standard input, standard output, and standard error**, which are files that represent the input to a program, the normal output, and the error output.

m) **Redirection** uses ">", ">>", ">&", ">>&", and "<" to send information somewhere other than the standard location. E.g., `ls -l > myFile` will send the output of `ls -l` to "myFile" instead of to the screen. If "myFile" already exists, the old contents will be erased first. If you use `ls -l >> myFile`, then it will work the same if "myFile" doesn't exists, but it will ***append*** if it already exists. If an error occurs, the error message still goes to the screen; but if the "&" versions are used, even the error messages go to the file.

   Using the "upper" program from above, I could enter my lines of text into a file called "myText.txt" and run `upper < myText.txt` to have all the upper case lines output to the screen or, to have the output also go to a file, I could run `upper <myText.txt > myUpper.txt`.

   Note that a quick way to create a file with two lines is:

```
echo This is line one > two.txt
echo This is the second line >> two.txt
```

n) ***File manipulation commands***

i) `cp myFile myNewfile` **makes a copy** of "myFile" under the name "myNewFile". Either argument can be a full PATH to a file in a directory other than the current directory. `cp myFile myDir/` makes a copy of "myFile" under the same name in "myDir". Any of the three methods (root, home, current) can be used to specify the directory. It is optional, but much safer to end the directory name with a slash to be sure you are not overwriting an ordinary file. You can (should) use the option "`-i`" if you want the shell to ask you to confirm if you are overwriting a file.

As an exercise, create a file called "hello.txt" with some text in it and print the text.

ii) `mv myFile myNewName` can be used to ***rename or move a file*** or directory.

As an exercise, create a folder named "newFolder". Move "hello.txt" into the folder, verify that it is no longer in the working directory, and print the text.

iii) `rm myFile` is used to delete a file (there is no recycle bin and no undo!). As an exercise, remove "hello.txt" and remove "newFolder".

o) `man myProgram` gets help on a program. Use `apropos keyword` if you don't know the exact name of the program you want. As an exercise, check how "rmdir" works and find what programs work with zip files.

p) Several ***utility programs*** are worth knowing

i) `wc fileName` does a ***word count***, returning the number of lines, words and characters in the file. Try `wc upper.c`.

ii) `sort fileName` returns the lines of the file ***sorted alphabetically***, ignoring initial blanks. Options include "`-r`" for reverse sort and "`-u`" for unique (removes duplicates). Play with `sort` using "upper.c".

iii) `head fileName` returns the ***first 10 lines of the file***. `tail fileName` returns the ***last 10 lines***. The option "`-n#`" can be used to change from 10 to another number. Try `head -n3 upper.c`.

iv) `cut -d@ -f #s fileName` ***cuts each line at the chosen delimiter*** (in the "@" position) and returns the "#s" elements only. The element choice can be a single "field number", a comma-separated list of field numbers of a dash-separated range of field numbers. Other functionality can be seen using `man cut`. Download "test.csv" from my "602/data" website and try `cut -d, -f 2,4 test.csv`.

v) `grep 're' fileName` outputs the lines of the file that match the regular expression. The repetition characters (*, +, ?) and some other special characters require backslash escapes to take on their usual regular expression meanings. For other than simple matching, egrep (extended grep) is more full-featured.

vi) `egrep 're' fileName` outputs the lines of the file that match the extended regular expression. Repetition characters are not escaped and {from, to} repetition counts work. Options include "-r" to output non-matches and "-i" to make the matching case insensitive.

Try `egrep "(<[a-z]+[.]h|r.{2}k)" upper.c`

vii) **sed** is the ***stream editor***. One main use is text substitution. The syntax is `sed –e 's/oldText/newText/g' fileName` which "g"lobally (i.e., repeatedly) "s"ubstitutes the "oldText" with the "newText", possible using regular expressions.

E.g., to change pyTestV3 into code for Python 2.x use:

```
sed -e 's/[()]/ /g' pyTestV3
```

You can use output redirection to put the results in a file.

viii) **`touch filename`** creates "filename" if it does not exist; otherwise it sets its last modified date and time to right now.

q) A **pipe** takes ***the standard output of program and sends it to the standard input of another program***. This can be very powerful!!

```
wc upper.c
grep '[^ ]' upper.c | wc
sed -e 's/^[ \t]*//' upper.c
sed -e 's/^[ \t]*//' upper.c | grep '[a-zA-Z]' |
     cut -d' ' -f1 | grep -v '[#*]' | sort –u
```

Pipes can be combined with redirection, e.g.,

```
sed –e 's/^[ \t]*//' < upper.c
grep '[^ ]' upper.c | wc > count.txt
```

r) The process management of UNIX allows you to run multiple jobs at the same time. To ***see your current jobs*** type `jobs`. You can kill a job (e.g., one with an infinite loop) with `kill %#` where "#" is the job number. You can get more ***information on all of the processes that you are running*** (including your shell) using `ps –f`. This shows the user id ("UID"), the process id ("PID"), the parents process's id ("PPID"), the current CPU usage, the start time "STIME", the terminal associated with the process "TTY", the total CPU usage time for the process ("TIME"), and the name of the process ("CMD").

```
UID        PID  PPID  C STIME TTY          TIME CMD
hseltman 28674 28673  0 08:05 pts/1    00:00:00 -bash
hseltman 29490 28674  0 16:18 pts/1    00:00:00 bash
hseltman 29527 29490  0 16:27 pts/1    00:00:00 ps –f
```

You can kill a process using its PID. To be sure it dies, use, e.g., `kill -KILL 29490`. Use `top` to get a dynamic listing of all processes.

Helpful hint: If you forget to use "&" when starting a program, you can type control-Z to suspend the program and get a command prompt, and then use `jobs` to see which job you suspended [normally #1], and then use `bg %1` to put job 1 "in the background" where it will run alongside of your shell.