Intro to DeterLab and Unix

Created by: Peter A. H. Peterson, Dr. Peter Reiher, UCLA. {pahp, reiher}@cs.ucla.edu

Contents

Overview

The "intro" lab is an introduction to DeterLab and some basic UNIX utilities.

The lab itself has four parts:

1. Obtain a DeterLab account and then create and run your intro experiment.
2. Use the command line to find five hidden JPG images.
3. Use the Internet to find answers to a few simple questions.
4. Make a "tar ball" of your answers and the images, and submit it to your instructor.

Finally, in addition to completing the lab, we recommend you spend some additional time playing around with utilities and reading `man` pages if you are unsure of your UNIX or command-line skills. You are welcome to use DeterLab for this purpose.

Required Reading

DeterLab Instructions

In order to complete this lab, you'll need to follow the instructions for for creating and running experiments. You'll also need to learn a few things about the Unix command line (if you don't already know them).

In this section, we will briefly discuss how to use the tools you *may* need to complete these labs. Most of these tools are very well documented. Your first reference should be the `man` page for the tool, usually accessed by executing `man program` on the command line. However, these tools are so fundamental to the command line that many of them have extensive documentation, tips, and tutorials available online. Google is your friend.

Secure Shell and Copy

SSH is short for Secure Shell, an encrypted, remote, command line terminal client. SCP is short for Secure Copy, and is used to transfer files from host to host using encrypted channels. Since you will use SSH and SCP to interact with DeterLab, your first step should be to read DeterLab's introduction to SSH and SCP for students.

the Unix shell

The Unix shell is the precursor to almost all modern "command line interfaces". When you ssh into DeterLab, you'll be using a "shell." Because similar interfaces are so common across many systems, we'll start by providing a translation key for common activities:

- change directories: `cd [directory]]`
- list files in directory: `ls [directory]`
- move/rename file: `mv oldfile newfile`
- copy a file: `cp oldfile newfile`
- delete a file: `rm file`
- remove an empty directory: `rmdir directory`
- create an empty directory: `mkdir directory`
- read a file: *see below*
- kill a process: `^C` (control-C)
- stop a process: `^Z` (control-Z)
- restart a stopped process: `fg`
- pause the terminal output: `^S` (control-S)
- resume the terminal output: `^Q` (control-Q)

... and many more. Most of these commands also have additional options. See `man command_name` for more information.

While this serves as a basic introduction, the shell's advanced features are too complicated to describe here.

For more information...

There are many good resources online for learning about the Unix shell:

- Introduction to Unix

- Learning the Shell
- An Introduction to the Unix Shell by S. R. Bourne

If you're interested in learning more, a great book on the subject is O'Reilly Media's *Learning the bash Shell* by Cameron Newham and Bill Rosenblatt -- although it is much more advanced than you will need for the typical exercises on DeterLab.

the Unix manual: the hitchiker's guide to Unix

`man`, the Unix manual, is where all the "help files" live in Unix. Practically every command has a helpful man page, which you can access by typing `man command_name`. It should be your first stop when looking for info about how tools work.

A few handy key strokes for perusing manpages:

- space goes forward a page
- return goes forward a line
- b goes back a page
- / searches forwards
- ? searches backwards
- q quits

Learning to read man pages is a skill. First, spend some time getting to know the general format: a one-line description, followed by a syntax synopsis (items in square brackets are optional), longer description, options, other arguments, various other documentation, optional examples, and references. If you're looking for a command-line switch to do something, chances are it should be in the first few sections of the manpage. Or, skip to the end and see if any examples do what you want. In any case, practice with "/" and "?" to search for text.

For example, suppose you were reading the manpage for `find` and you wanted to know how to keep `find` from looking in other filesystems connected to the machine (such as network drives). The manpage for `find` is over 1,000 lines! Who has time to read the whole thing? However, if you use the "/" forward search key and enter "filesystem," the second match is for the command-line switch to do just that (`-mount`).

Other sources of information

If a tool doesn't have a man page, or you'd like some different information, the following usually works:

```
$ command_name -h (or --help)
```

This will often print out a useful page of information. If there is too much output to read, try using one of the pagers (e.g. `more` or `less`) described below.

If you still need more help, search online. Or, email your instructor, TA, or friendly neighborhood Unix guru.

sudo: superuser do

sudo is a Unix command meaning "superuser do" and gives unprivileged users access to privileged commands. If you try to do something and it says `Permission denied`, try again -- this time, put `sudo` in front of the command.

> `sudo` is essentially a setuid-root program that interprets its own expressive ACL to determine which actions should be allowed. If you don't know what that means now, don't worry -- you will later.

The most simple use of `sudo` is to give a user (such as the primary user of a Linux system) full access to root abilities. In order to exercise those abilities, the user executes the command through sudo like this:

```
$ sudo shutdown -r now
Password: [user's password]

This system is going down for a reboot NOW!
Connection closed by foreign host.
```

This command (to reboot a Linux server immediately) typically requires root access. Without `sudo`, a user would have to become root using the command `su` (switch user):

```
$ su -
Password: [root's password]
[ROOT $] shutdown -r now
This machine is going down for a reboot NOW!!!
```

But this requires that the unprivileged user knows root's password, which undermines any concept of real privilege or security. Instead, `sudo` requires the user to enter *their own password*, which serves to prove that their terminal has not been taken over. The `sudo` ACL is then interpreted to ensure that the user has the permission to execute the command entered, at which point it is executed as root.

Typically, the user enters their password the first time they use `sudo` in a session (it actually "times out" in around 15 minutes). Sometimes, systems are set up (like DeterLab's experimental nodes) where `sudo` does not require a password. This can be dangerous; one of the nice features of the password prompt is to make sure that you really mean to execute something as root.

`sudo` can lock down individual commands; this is necessary for real security. For example, a user can be given access to run a particular program as root but no root access to any other actions. Due to the expressiveness of the ACL, it is very easy to inadvertently create an insecure ACL allowing a user unintended access through `sudo`. For example, on DeterLab you can enter `sudo su -` to assume `root` without knowing `root`'s password. You can also enter `sudo passwd root` and change `root`'s password. In the case of DeterLab this is by design -- but obviously in other environments it would represent the worst kind of security breach, because once an attacker has acquired root access there is *no way to know the system is secure* without wiping and reinstalling or validating every single piece of software on the system. (Why?) Therefore it is important that administrators make sure their `sudo` ACLs are well written and secure.

> ⚠ If an attacker has root access, he or she can change literally anything that is changeable on the system, including modifying the kernel, executables, compilers, system tools, and so on. Common modifications

> include changing the tools that would normally be used to detect modifications! Because of this, it is extremely difficult to verify that a system has been "secured" following an attack. As a result, the "best practice" is to wipe the system to the "bare metal," including reformatting the hard drive and reinstalling the operating system and data from trusted media.

There isn't space here to go into the full `sudo` ACL (which is located at `/etc/sudoers`) but there is a `sudo` manpage, many online tutorials, and even a `sudoers` manpage for the ACL file, featuring a complete syntax description in EBNF. Enjoy that.

Read and search in files

cat, less, tail, head, and grep

cat

`cat` (short for concatenate) opens a file and prints it to standard out (which is typically your console). `cat` doesn't have any "brakes" -- it will flood your terminal -- but it is indispensible for sending data into other Unix applications. The command:

```
$ sudo cat /var/log/messages
```

... will print the file /var/log/messages to screen as fast as your connection will allow. `cat` terminates at the end of the file. You can also quit `cat` by pressing `^C` (Control-C).

Most of the time, however, you want to control how much of a file you see. The following tools help you do just that.

less

`less` is the better replacement for the Unix file pager `more`. To use `less`, enter:

```
$ less /var/log/messages
```

... or

```
$ cat /var/log/messages | less
```

And you will be greeted by the top of the system log. To move up and down in the file, you can use the arrow keys, or page up, page down, home, and end. You can also search within the loaded file using the / (search forward) and ? (search backward) command, like this:

```
...
xxx.105.166.xxx - - [02/Sep/2007:07:15:32 -0700] "GET /foo/SomePage
HTTP/1.1" 200 15289
xxx.105.166.xxx - - [02/Sep/2007:07:17:23 -0700] "GET /foo/ HTTP/1.1" 200
16557
/SomePage<enter>
```

Note the bottom line, `/SomePage<enter>`. When you press "/" (the search forward key), less will print it at the bottom, and wait for you to enter a search string. When you're finished, press enter. This will jump to the first and highlight all occurances of the string "SomePage". To see the next result, press "/" again and hit enter. In this way, you can cycle through all occurrences of a string in a text file. The command "?" works exactly like "/" but searches backwards. Both ? and / accept regular expressions (also known as regexes) in addition to normal strings -- if you know regexes you can create vastly more expressive search patterns.

Hit q to quit `less`.

tail and head

`tail` and `head` respectively print out the last 10 and first 10 lines of their input file. Typically, `tail` is used to check the end of a file, but it is also very commonly used to "watch" a log file. Using the command:

```
$ sudo tail -f /var/log/messages
```

... you can watch the messages file grow. ^C quits.

grep

`grep` is what makes `cat` useful in this context. `grep` is a filter that uses patterns (including regexes) to filter lines of input. For example, given the snippet of the messages file from before, if a user "pipes" the output of `cat` into `grep` and filters for "xxx.55.121.xxx" like this:

```
$ cat /var/log/messages | grep xxx.55.121.xxx
```

... she will see only lines matching xxx.55.121.xxx:

```
...
Sep  5 21:49:08 localhost postfix/smtpd[19090]: connect from
unknown[xxx.55.121.xxx]
Sep  5 21:49:10 localhost postfix/smtpd[19090]: lost connection after
CONNECT from unknown[xxx.55.121.xxx]
Sep  5 21:49:10 localhost postfix/smtpd[19090]: disconnect from
unknown[xxx.55.121.xxx]
...
```

If there is still too much output, just pipe the output from `grep` into `less`, like this:

```
$ cat /var/log/messages | grep kernel | less
```

... and now you can use the features of `less` to examine your result.

As an alternative, you can use *command line redirection* to send the output to a file like so:

```
$ cat /var/log/messages | grep kernel > kernel_grep.txt
```

... You can then use `less` on the file `kernel_grep.txt` you just created.

`grep` has many advanced features, such as negation (`grep -v somestring`). For more information see `grep`'s manpage.

Redirecting Output to a File

💡 You can make a command put its output into a file in any location like this:

```
$ somecommand > top_secret/output.txt
```

You could also do this with copy and paste via `screen`, but output redirection is nicer for most purposes.

find, xargs, and locate: find files

Users of more "user friendly" operating systems such as Windows and OS X are spoiled when it comes to finding local files, because while the graphical tools like Windows find, Apple's Spotlight Search, and Google Desktop are fast and easy to use, they are generally not nearly as flexible or expressive as the standard Unix utilities for finding files, `find`, `xargs`, and/or `locate`.

find -- find files on the system

`find` can be used to search for files of various names and sizes, various modification times, access permissions, and much, much more. However, the syntax for `find` is a black art into which most of its users are barely initiated. We'll discuss the basics here so you can use it. If you want to know more, read the manpage or look online.

The basic command format is "`find [path [expression]`", where 'path' is the directory to start searching in and `expression` is some compound expression made up of *options*, *tests*, *actions*, and *operators*. The expression modifies the search behavior: *options* specify things like how many levels deep to search, whether to follow symlinks, whether to traverse filesystem boundaries, etc. *Tests* specify conditions like matches on the filename, modification date, size, etc. *Actions* can be defined to delete matching files, print the files, or execute arbitrary commands. (The default action is to print the name of any match.) *Operators* are logical operators for combining multiple expressions. Expressions limit results. Accordingly, no expression at all will "match" everything and the default action will print the relative paths of all files that `find` encouters.

You usually don't want to list every file in a subtree. In this case you may want to limit the search with an expression. An expression begins with the first of several expression options that begin with a hyphen (such as `-name` or `-mtime`) or other special characters. The expression can also specify actions to take on any matching files (such as to delete them). Expressions can become very complicated. And like any complicated machine, the more complicated the expression, the more likely it is that `find` will not do exactly what you want. If you need to create expressions beyond the complexity addressed here, please see `man find` or a tutorial online and try examples on your own.

Here are a few simple examples to get you started:

"Find all files ending with .txt, starting in this directory. Use `head` to show me only the first 5 results."

```
$ find . -name "*.txt" | head -n 5
```

"Find all files ending with "~" or ".swp" (common tempfile designations), starting in the subdirectory `public_html`." To do this, we'll use the OR operator `-o`.

```
$ find public_html -name "*.swp" -o -name "*~"
```

"Find all files on the root filesystem not modified in 6 years (2190 days). Start in the root directory (/), and do not descend into other filesystems (such as NFS directories). Use head to only show the first 5 results."

```
$ find / -mount -mtime 2190 | head -n 5
```

Finally, here's an example from `man find`:

```
$ find /    \( -perm -4000 -fprintf /root/suid.txt ’%#m %u %p\n’ \) , \
             \( -size +100M -fprintf /root/big.txt ’%-10s %p\n’ \)
```

This translates to: "Traverse the filesystem just once, listing setuid files and directories into /root/suid.txt and large files into /root/big.txt." (`man find`)

> 💡 The backslashes ("\") in the above example are called "escapes." Backslash escaping tells the shell either to treat the escaped character literally -- not as a a special character -- or vice versa (that a regular character has a special meaning). Above, the parentheses are escaped because they are input for `find`. If they were not escaped, the shell would interpret them in its own way, because parentheses are special for the shell. Similarly, a line break created by pressing enter signifies the end of a command -- unless it is escaped. Here, the line break (visible as the backslash at the end of a line) is escaped to tell the shell that the command is continuing on the next line. Finally, an escaped "n" character, (`\n`) represents a line break in printf-style format strings.
>
> Be careful with advanced uses of `find`, especially if you are deleting files. As you might imagine, unexpected things often happen when shell escapes, quoting and expansion are involved.

Finally, because find performs a depth-first search over potentially the entire filesystem tree, it can take a long time to run. You may have better luck with `locate` depending on what you're trying to do. (See below.)

xargs -- build command lines from standard input

In the Unix environment, the output of one program can be easily redirected (or "piped") into the input of another program. `xargs` allows you to use the output of one program as *arguments* to another process. This is very useful in conjunction with tools like `find`, `cat`, and `grep`. As with everything in the LabTools document, this is only a brief introduction to `xargs` -- for more information, please see `man xargs`.

The most common use of `xargs` is to take the output from a command and use it as a parameter in a different command. For example:

"Run `strings` on every file on the system that ends with ".pwd" (case insensitive) and sort the results ASCIIbetically."

```
$ find / -iname "*.pwd" | xargs strings $1 | sort
```

The output of `find` is used by `xargs` to construct the `strings` command. After this, the output of `strings` is put into a different order by `sort`.

Here's another example:

"Delete the first four files ending in ".swp" or "~" that have not been modified in 24 days starting in this directory."

```
find . -mount -mtime +24 \( -name "*.swp" -o -name "*~" \) | head -n 4 |
xargs rm $1
```

Note the use of `head` to limit the action. Without `head -n 4`, `xargs` would have removed every matching file.

locate -- a faster way to search

`find` does not keep a database of information to use; every time you run a new invocation of `find`, it traverses the entire directory structure looking for matches. This allows you to search the current state of the system, but also lacks the advantages that pre-indexing and caching can provide. `locate` and its helper command `updatedb` are more like Google Desktop than Unix `find` in this sense.

Before using `locate`, `updatedb` must be run as root to index the entire disk, recording it in a database. Then, the `locate` command is used with a few simple options to match filenames. This can be *much* faster than using `find`, especially if you don't know exactly what you're looking for or where it might be. As always, see `man locate` for more information. To update the locate database, run:

```
$ sudo updatedb
```

`sudo` is used to make sure that you have permission to read the entire disk.

The following examples assume that updatedb has been run as root recently:

"Find all files ending in '~' or '.swp'."

```
$ sudo locate *~ *.swp
```

"Find all files ending in '~' or '.swp' and delete them."

```
$ sudo locate *~ *.swp | xargs rm $1
```

`locate` will complete the search for the first term and then the second term. So an example like this:

"Find all files ending in '~' or '.swp' and delete the first 10 results."

```
$ sudo locate *~ *.swp | head | xargs rm $1
```

Will only delete ".swp" files if there were *less* than 10 "~" files (since the "~" files are returned first).

`sudo` is used in the locate commands to make sure that you are able to access all of the `locate` database.

tar

tar -- Tape Archiver

tar is an ancient program, reflected in the meaning of its name -- tape archiver -- from when backups were primarily performed on magnetic tape.

The tar manpage includes a lot of information, but I'll just give some example usage:

To create a `tar` archive (also called a "tar file" or "tarball") of an entire directory, execute:

```
$ tar cvzf somedir.tar.gz somedir
```

In the above example, there are a number of switches:

- c -- create archive
- v -- be verbose
- z -- compress (adds gzip (gz) compression)
- f -- tells tar that you're going to provide the filename next

In this case, `somedir.tar.gz` (the first parameter) is the file to be created, and `somedir` (second parameter) is the directory being compressed.

To decompress a tar file, do something like this:

```
$ tar xvzf somedir.tar.gz
```

In this example, the x stands for extract; the other switches are the same.

screen: terminal multiplexer and manager

screen is a terminal multiplexer with many features. Typically, when a user disconnects from a system, all their running processes terminate -- `screen` allows users to keep those programs running in such a way that the user can reconnect to them. Screen also allows users to have multiple shell sessions open in a single terminal application window.

The most immediately obvious benefits of `screen` are:

- disconnecting from a system but leaving your programs running in such a way that you can reconnect to them
- share a screen between two users
- cut and paste
- multiple terminals "inside" one terminal window

`screen` functions as a very simple window manager for terminal sessions. As such, all programs are run *inside* `screen`, which itself runs inside one terminal window (such as PuTTY, Gnome Terminal, or Terminal.app). Control sequences (starting with Control-a) are used to issue commands to `screen`.

For example, to start a large software compilation and detach your terminal, one could enter these commands:

- screen *(this starts the program screen and a terminal inside of it)*
    - $ make bigsoftwareproject (this will take a while)
    - Control-a, d *(detach this screen)*
- exit *(log out of the system)*

Later, the user can log back in to the system and type:

- screen -r -d *(detach and pick up a screen session)*
- ... Now the user is back where they left off, and the software is compiling in the background.
    - Control-a c *(create a new window)*
    - $ less somefile.txt *(The user looks at a file inside less)*
    - Control-a 0 *(The user switches back to screen 0 where "bigsoftwareproject" is still building)*
    - ... (It's still building...)
    - Control-a 1 (The user switches back to screen 1 where they are reading somefile.txt)
    - ... etc.

For example, a user can keep an IRC window, mail client, software build, top system monitor, iptraf monitor, and more running on a system as long as the system stays up. The user can access the `screen` session, interact, and disconnect from the session with the ability to log back in to the system and pick up where he or she left off.

`screen` also allows you to copy and paste in between windows and scroll back. In order to use this functionality, enter `Control-a ESC`, followed by the arrow keys to navigate the cursor. Press ENTER to make the first mark, move the cursor, and press ENTER again to select and copy the text. To paste it at the cursor location, press `Control-a ]`. Because screen has a large scrollbuffer, the copy function can be used to scroll back long distances, which is useful for reviewing past information. To leave copy mode without selecting any text, hit ENTER twice.

top and ps: process lists

Top is an interactive process list and can be used to monitor aspects of the current system state. Top looks like this:

```
$ top
top - 21:18:50 up 15 days, 31 min,  6 users,  load average: 0.35, 0.22,
0.17
Tasks: 134 total,   1 running, 133 sleeping,   0 stopped,   0 zombie
Cpu(s):  3.3%us,  0.3%sy,  0.0%ni, 96.3%id,  0.0%wa,  0.0%hi,  0.0%si,
0.0%st
Mem:   1034856k total,   998196k used,    36660k free,   228012k buffers
Swap:  1965584k total,    65512k used,  1900072k free,   388076k cached
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
21415 root      15   0  185m 106m  14m S  3.3 10.6   9:48.08 Xorg
18458 pedro     25  10  2320 1184  880 R  0.3  0.1   0:00.02 top
22544 pedro     15   0  109m  31m  11m S  0.3  3.1   0:07.90 gnome-
terminal
    1 root      15   0  2908  508  456 S  0.0  0.0   0:01.38 init
    2 root      34  19     0    0    0 S  0.0  0.0   0:06.73 ksoftirqd/0
    3 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
...
```

Top updates roughly every two seconds. The rows can be sorted in multiple ways, but probably the most useful commands to know are 'P' to sort by processor use and 'M' to sort by memory use. These commands are case sensitive. Hit 'q' to quit top.

If you need more comprehensive information, try using the `ps` command. It's not pretty like top, but it's thorough. Try running something like this:

```
$ ps aux
```

or...

```
$ sudo ps aux
```

> If there's too much information there for you, try using one of the previous shell tools (e.g., `sudo ps aux | less`) to manage it or output it to a file.

elinks: terminal based web-client

ELinks is a text-only web browser that can handle tables, frames, and ssl. To use Elinks, enter a command like one of these:

```
$ elinks http://example.com/
$ elinks http://10.1.1.2/foo.html
$ elinks index.html
```

ELinks is very feature-rich, with many hotkeys and a GUI-style dropdown menu you can access with the escape key. Navigate the menus and webpages with the arrow keys, pressing right arrow or enter to open a link, or left arrow to go backwards.

editors -- edit hexadecimal and ASCII: edit text and binary files

There are many traditional editors, ranging from the very powerful to the simple and even the incomprehensible. You may already have your favorite editor, but if not, feel free to choose from this list:

nano - the easiest editor for beginners

`nano` : Homepage : Wikipedia : Documentation

`nano` was originally written to be a Free Software version of `pico`, the editor included in the once-popular email client, Pine. `nano` is very similar to familiar text editors such as MS-DOS's `edit` and `notepad`. As such, it is very easy to use, but lacks most of the advanced features in Emacs and vi that make those editors popular with programmers. **How to install nano**

If not already installed, you can install `nano` by typing `sudo apt update; sudo apt install nano -y`.

Quitting nano

Hold `CTRL` key and press "x". You can see all commands at the bottom of the screen. "^" means "CTRL".

vim

`vim` : Homepage : Wikipedia : Vimdoc

Vim is "vi-improved" and is based on `vi`, written by Bill Joy in 1976. `vi` is a modal editor (which means you are typically either in insert or command mode but never both simultaneously). To enter insert mode for entering text, press `i` or the `Insert` key on your computer. (`vim` will enter Replace mode (i.e. overwrite) if you press `Insert` twice). To leave insert mode and enter command mode, press `Escape`.

`vi` is part of the POSIX standard for Unix, so some version of `vi` (e.g., `vi`, `nvi`, `elvis`, `vim` ...) should be available on all Unix-like systems everywhere. Vim has greatly extended features such as split screens, syntax highlighting, and much more. `vi` is often lauded for doing one thing and doing it well -- on the other hand, it is sometimes criticized for this limitation, and many users dislike modal editing. New users sometimes find `vi` confusing.

How to install vim

If not already installed, you can install `vim` by typing `sudo apt update; sudo apt install vim -y`. **How to quit vim**

To quit vim, hit `escape` (to enter command mode), then enter

```
:wq
```

This stands for write and quit -- just `:w` writes without quitting (a.k.a "saving"), `:q` quits a blank document, and `:q!` quits a document with unsaved changes. Finally, `:w!` writes a "write protected" file that you have the permission to change.

emacs

`emacs` : Homepage : Wikipedia: Tutorial

Emacs is the Pepsi to `vi`'s Coke. Originally written by Richard M. Stallman at MIT, Emacs is first a text editor, but it is much more. Emacs is incredibly extensible; it features a built in version of the Lisp language, and many external libraries and applications exist to add functionality. With the standard and extended features, Emacs can be and do so many things that some people think of Emacs almost as its own operating system. Advanced functionality includes: gdb debugger interface, diff, gnus email client, web browser, file browser, tetris, and pong! Emacs is appreciated for its power and unparalleled extensibility, but is often criticized for being bloated or for trying to do "too much". New users sometimes find Emacs confusing.

How to install emacs

If not already installed, you can install `emacs` by typing `sudo apt update; sudo apt install emacs -y`.

How to quit emacs

You can quit emacs by holding `CTRL` key and pressing `X` and then `C`. If you have unsaved changes, you will be prompted to save them. Type "y" to save or "n" to quit without saving.

joe

`joe` : Homepage : Wikipedia : Tips

Joe's own editor is designed to be easy to use, and is very common in Linux distributions. It is similar to `nano` in its simplified "`notepad`-like" interface, but has somewhat more features than `nano`.

How to install joe

If not already installed, you can install `joe` by typing `sudo apt update; sudo apt install joe -y`.

Quitting joe

Hold `CTRL` key and press "KX".

ed

`ed` : Homepage : Wikipedia : Notes

`ed` is the original standard Unix text editor, written by Ken Thompson, and it is in some ways a primitive ancestor of `ex`, `vi`, `awk`, `sed`, and Perl. `ed` is terse *in the extreme*, a design feature that was logical in the days of 110 baud [slow!] terminals and remote connections, but which today seems alien, unfriendly, and unbelievably obtuse. Nonetheless, `ed` is even more universal than `vi` and serves as an interesting paleontological counterpoint in any discussion of text editors. **How to install ed**

If not already installed, you can install `ed` by typing `sudo apt update; sudo apt install ed -y`.

Quitting ed

Press "q" to quit.

See each editor's `man` page for more information.

Introduction

This simple lab needs no introduction. Just follow the instructions and submit the resulting tarball to your instructor.

Assignment Instructions

Assignment Instructions

Setup

1. If you don't have an account, follow the instructions here.
2. Create an instance of this exercise by following the instructions here, using intro as Lab name. Your topology will look like below:

3. After setting up the lab, access your `intro` node.

Tasks

1. Treasure Hunt

1. On the `intro` node there are 5 JPEG files whose names contain the word "intro" **in some form**. Find all five files.
   - Recall that UNIX is case sensitive.
2. Make a directory in your home directory called `top_secret`.
3. Move the 5 files into `top_secret`
4. Use a text editor to create a plain text file in `top_secret` called `answers.txt` that includes the original locations of the 5 files and a brief but unambiguous description (e.g., the title) of the image.

> 💡 To describe the image, you need to view the image. This involves using Secure Copy (`scp`) to transfer the files to your local machine. See introduction to SSH for more information.

2. Information Hunt

It's important to be able to go online and find the answers to technical questions. In order to get you figuring out how to find answers, we'd like you to answer the following questions. Put the answers to these questions in your `answers.txt` file in the `top_secret` directory.

1. 1 sentence: What goes in the `/var` directory on a UNIX computer?
2. 1 sentence: What is the `/dev` directory for on a UNIX computer?
3. On your experimental node, find out how large the disks are and how much space is free. Put this information in a separate file called `top_secret/diskfree.txt`. (See the infobox on command redirection for an easy way to do this.)
4. On your experimental node, find out the "vendor id" of the experimental node's CPU model.
   - Hint: there is a dynamic file on the system that includes this information.

3. Wrap it up!

Collect your answers and the images you found and submit them to your instructor.

1. Make sure your personal information (and any other information requested by your instructor) is included in `answers.txt`:
   - Your name
   - Your email address
   - Your DETER username
2. Make a gzipped tarball called `username-intro.tar.gz` containing the `top_secret` directory
   - This should include all 5 images, `diskfree.txt` and your `answers.txt`

  - Example: `tar cvzf 1234567890-intro.tar.gz top_secret`

What can go wrong

There's not much to go wrong here. However, when you're looking for the image files, remember that Unix is case-sensitive -- "I" and "i" are not the same. Also, remember that sometimes, different extensions can refer to the same type of file.

Submission Instructions

Submit your tarball to your instructor.