Unix Terminal SkillsStatistics 506

What is Unix?

Unix is an operating system that underpins most modern operating systems. It was developed in the 1960's and 1970's at Bell Labs, a famous research center owned by AT&T. It features an entirely **text-based user interface**, requiring only a keyboard to interact with it (though mice are supported on modern unix systems if desired.)

A **kernel** is the central part of an operating system that controls all operating aspects of the system, the most crucial being the link between hardware and software. Most users will never interact directly with the kernel, instead interacting with **processes**. (An "app" is a **graphical user interface** to a process.)

Unix's success encouraged the development of two other important operating systems, Linux and BSD.

Linux was released in 1991 as an **open source** alternative not owned by AT&T, meaning any one was free to view, modify, or re-distribute the **source code**, the code in which Linux was written. This freedom has lead to innumerable other operating systems based on Linux, most notably Android, the operating system at the heart of Google Android phones and devices. Additionally, the vast majority of remote servers in which researchers carry out analysis are some variation of Linux.

BSD is a [**fork**] (or modified copy) of Unix developed in the 1970's at UC Berkeley. Like Linux, it too was released under open source. BSD has itself led to other systems such as Orbis OS, the operating system that drives the Sony PlayStation, but most notably, **DarwinOS** is based upon BSD - and **DarwinOS** is what all of Apple's operating systems (macOS, iOS, ipadOS, etc) are based on.

The distinction between Unix, Linux and BSD

Unix, Linux and BSD are all distinct operating systems (not even including all of their derivatives and forks), but for the vast majority of users in the vast majority of situations, *they can be treated as indistinguishable*. Processes and software that work on one of the operating systems

will almost always work on the other operating systems without any modification from the user.

We call all of these systems **Unix-based systems** to imply this.

How does Windows fit in?

It doesn't. **Microsoft Windows** is built upon the **MS-DOS** operating system kernel, which itself is based on the **DOS** operating system kernel, which came from IBM.

In the past, this divide meant that Windows users who wished to take advantage of Unix-based software either were out of luck, or had to use a third-party tool like PuTTY which emulated a Unix-like interface on top of Windows. This was clunky and annoying.

More recently, Microsoft released **WSL**, the Windows Subsystem for Linux. This allows you to interface with your Windows-based computer using a Unix-based interface. I know little about this, not having used Windows for anything serious in almost 20 years, but if you have a Windows computer, you can investigate whether you want to use WSL: https://learn.microsoft.com/en-us/windows/wsl/about.

Command-line interface

While the overwhelming majority of interactions users of computing resources have these days is via a graphical user interface, the alternative of a command-line is still used heavily in research environments. While GUI interfaces (such as RStudio) have their uses, they require substantially more computing resources (you can imagine how much harder it is to display a complex graphical display versus some words on a black screen). This is especially true when working on a cluster such as Great Lakes where you want to devote all power towards your large job.

If you are using a Mac or Linux system, as these are Unix-based systems, you have access to a command-line interface. On a Mac, launch the "Terminal" app. There are different versions available on different Linux distributions, please reach out to me if you can't find a command-line application.

The basics of interacting with the command line

There are a large number of tutorials online about working with the command line. We will briefly discuss these, but I encourage you to work through some of these tutorials on your own time if this is new to you. Here are some I've not thoroughly vetted but look promising:

- Very basic: https://medium.com/swlh/how-to-use-the-command-line-interface-cli-9c8b70e568e
- UM ITS tutorials: https://documentation.its.umich.edu/node/295/ and https://documentation.its.umich.edu/node/239/ and https://documentation.its.umich.edu/node/297/
- Specific to the Ubuntu variation of Linux, but generally useful: https://ubuntu.com/tutorials/command-line-for-beginners
- Some of the other sections of this tutorial are useful but more advanced: https://ryanstutorials.net/linuxtutorial/navigation.php
- An "intermediate" tutorial: https://jayconrod.com/posts/103/intermediate-linux-command-line-tutorial

A quick summary of commands

You can use the following commands to navigate and interact with the file tree:

- `pwd` (print the current or **working** directory)
- `cd` (change directories)
- `ls` (list files), `ls -a`, `ls -l`
- `mkdir` (make directory), `mkdir -p`
- `rmdir` (remove directory)
- `rm` (remove a file), `rm -r`
- `cp` (copy a file or directory to another file or directory), `cp -r`

In working with the file system, it is helpful to know:

- `.` refers to the current directory. E.g. `cd .` does nothing!
- `..` refers to the parent directory, one step up the file tree. E.g. `cd ..` will move to the folder containing your working directory.
- `~` refers to your home directory. E.g. `cd ~` returns you to your home directory.
- Configuration files and others used by programs are often named as **hidden files**. The names of hidden files begin with a `.`. To see these files, use `ls -a`.
- Use *filename wildcards* to refer to groups of files matching specific patterns:
  - `*` matches any sequence of characters
  - `?` matches any single character.

Hierarchical file structure

Unix is a file-based operating system - everything is a **file**. Your files (obviously), but also executables (which launch processes or applications), data, and any configuration files.

The structure is **hierarchical**, that is, the **root** directory is `/`. There are a number of folders (and sometimes files) inside `/`, for example `/bin`, `/opt` or `/var`. You don't need to know about most of these as they contain almost entirely system files that you'll hopefully never need to interact with.

User files are usually stored in `/Users/<username>` on macOS, or `/home/<username>` on Linux systems. Your folder in here is known as your **home directory** and it is where you should store almost all of your personal files. When referencing your host directory, you can of course use the full path, e.g. `cd /home/<username>/my_files`, or you can use `~` as a shortcut as mentioned above: `cd ~/my_files`.

The shell

The interface to the unix-like operating system and its file structure is your **shell**, typically via a **command line interface**.

Several different shells are available on modern unix-like systems. The most commonly used shell is probably the [bash](<https://fishshell.com/) shell. Other> alternatives include [zsh](<https://en.wikipedia.org/wiki/Z_shell), which is a> more full-featured version of bash, and fish, which is a very modern alternative. I would recommend becoming familiar primarily with bash, as it is almost guaranteed to be on any remote server you work on. Zsh is slightly less likely to be found on remote servers; fish is extremely unlikely to be found.

There's an interesting discussion about the difference between "shell" and "terminal" if you're interested: https://unix.stackexchange.com/questions/4126/what-is-the-exact-difference-between-a-terminal-a-shell-a-tty-and-a-con

Interacting with the shell

You can use the following commands to navigate and interact with the file tree:

- `pwd` (print the current or **working** directory)
- `cd` (change directories)
- `ls` (list files), `ls -a` (list all files including hidden), `ls -l` (include additional information)
- `mkdir` (make directory), `mkdir -p` (recursively, make necessary subdirectories)
- `rmdir` (remove directory)
- `rm` (remove a file), `rm -r` (remove recursively, e.g. a folder and all subfiles)
- `cp` (copy a file or directory to another file or directory)
- `man` (help page for another command, e.g. `man mkdir`)

In working with the file system, it is helpful to know:

- `.` refers to the current directory. E.g. `cd .` does nothing!

- `..` refers to the parent directory, one step up the file tree. E.g. `cd ..` will move to the folder containing your working directory.
- `~` refers to your home directory. E.g. `cd ~` returns you to your home directory.
- Configuration files and others used by programs are often named as **hidden files**. The names of hidden files begin with a `.`. To see these files, use `ls -a`.
- Use *filename wildcards* to refer to groups of files matching specific patterns:
  - `*` matches any sequence of characters
  - `?` matches any single character.

Text editors

There are a lot of text editors available for editing files directly at the command line. If you are working locally, you can of course edit files in your favorite graphical editor, but for quick edits, it is usually faster to edit in the command line, and the graphical editor is usually not available to you on remote servers.

There are three primarily used editors:

- nano (some systems, notably macOS, have pico instead, which is a less-featured version of nano, but still sufficient for quick edits.)
- emacs
- vi / vim

If you are going to be spending any time at the command line, you should become familiar with one of these on your own. Nano/pico is the least featured but easiest, you can learn to use it in a matter of minutes. Vim is a modal editors which may appeal to you but has a decent learning curve. Once proficient its users are able to edit code extremely quickly. Emacs is by far the most powerful (it has been called "an operating system masquerading as a text editor") and the most extensible, but also comes with the steepest learning curve.

Connecting to remote servers

SSH (the Secure Shell Protocol) is the primary tool used to connect to remote servers. Once you establish an SSH connection to a server, you can work on the command line of that server.

At Michigan, there are two servers of note:

The "login" servers provide access to a unix environment. This should **not** be used for substantial computing, though you can do light work there.

```
ssh <username>@login.itd.umich.edu
```

The Great Lakes cluster as an alternative to the web interface:

```
ssh <username>@greatlakes.arc-ts.umich.edu
```

Don't forget to use `module load R` or similar when working on Great Lakes. The login servers generally don't have statistical software unless you install it yourself.

Preserving remote sessions via terminal multiplexer

If you are connected via SSH and you lose connection (either manually disconnecting, or via a power/internet issue), you lose the current SSH session. Your files are unaffected, but any code currently running (whether it be an active script, or just the fact that you perhaps have R open in an interactive session) get lost.

Terminal multiplexes help address this issue by preserving your session regardless of your connection status. There are of course a number of different tools, but I would recommend starting with screen, as it's available on most systems.

You start screen by calling `screen`. From this point, you are inside the screen session. You can manually "detach" it by key command "ctrl+a d". You can re-attach it when outside of screen by calling `screen -r`. If your session disconnects, `screen -r` should reattach it as well.

Submitting jobs to Great Lakes via SSH

As an alternative to the web interface for submitting jobs to Great Lakes, you can submit via the SSH connection. Once you've connected, you can process your jobs:

- `sbatch <path/to/slurm_script.sh>`: Submit a job
- `sq <username>`: View your jobs in queue
- `squeue`: View *all* jobs (very large output!)
- `my_accounts`: Shows your allocations
- `my_usage`: Shows all charges you've accrued