CHEG 667-013 – CHEMICAL ENGINEERING WITH COMPUTERS
Department of Chemical and Biomolecular Engineering
University of Delaware

Spring 2025

COMMAND LINE INTERFACE PART II

Key ideas today:

Continue exploring the Linux (or Unix) command line interface.

Key goals:

Learn how to manipulate and edit files, file lists, and input / output. Learn about the shell and some of its features, including scripting.

We continue on our quest to learn about the command line. There is a lot to learn and the commands are somewhat cryptic. The best way to get comfortable with the command line is to use it. Look around the system. You can't really break anything. You generally don't have the permissions to edit, move, or delete system files.

That said, remember: there is no undo when it comes to your own files. A good practice is to make backup copies of important files!

1 System information and managing processes

There are several useful programs for seeing who or what is running on a machine.

- `who` – display who is on the system
- `w` – display who is on the system and what they are doing
- `whoami` – display the current user ID
- `ps` – process status – information about processes on the system
- `top` – display sorted information about processes (like activity monitor)
- `uptime` – show how long system has been running

`ps` returns important information like the *process id*, a unique number assigned to every running process. The command has a few options, and a somewhat strange syntax. Most systems will interpret `ps aux` as a command that lists all running processes on a system.

Exercise 1: Try out each of the commands above!

Killing a process

If you have a process that seems to be hanging, like a program with a buggy infinite loop (maybe `top` shows a lot of CPU use and the fan is spinning loudly), you can terminate it with the `kill` command. You first have to find the process id using `ps`. Then type:

```
ef1j@snaut:~$ kill pid
```

2 Reading files

We've already used the `cat` command to print a file. Several other programs are useful for scanning through files. These include:

- `cat` – print a file
- `more` – page through a file one screen at a time
- `less` – "Opposite of more." (macOS `more` actually runs `less`.)
- `grep` – file pattern searcher

Exercise 2: Make a directory listing of the root file system with `ls -lR / > lsR_filesystem.txt`. Then page through the file using `less`. Move up and down. Can you find a particular file using a search?

Of the programs above, `grep` is a little different. It is used to find files with matching strings. It's basic use looks something like this:

```
ef1j@snaut:~/cheg667$ grep 'string' filename
```

which will look for `string` in the file `filename`.

More help with less

Incidentally, in addition to the `man` pages, many commands have built-in help. Try running `less --help`. This also shows us that some options are proceeded by a long dash. You might also try `less -?`.

3 Pipes and redirects

Pipes and redirects enable you to control the flow of information to and from processes.

- `|` – a pipe
- `>` – redirect output to a file (this will always overwrite the file)
- `>>` – redirect and *append* output to a file
- `<` – use a file as input

One of the design philosophies of Unix centers around providing small, focused program tools that can be used together. Pipes send the output of one program to the input of another. Try it! Type:

```
furst@snaut:~$ ps aux | less
```

The `ps aux` command will generate a long list of processes that scroll past quickly. By "piping" the command to `less`, it is easier to page through the output.

In Excercise 2, we redirected the output of `ls` to a file.

4 Editing files in the terminal

Most of the time, we can use an editor or development environment to write code. However, sometimes we need to write or edit a file in the terminal, including configuration files. There are a few options:[1]

- `vi` – "a programmers text editor"
- `nano` – a "small and friendly editor"
- `emacs` – Emacs is more than an editor.
- `ed` – line editor for Yoda-level Jedi editing.

Exercise 3: Using one of the editors above, type in the following short c program:

```
#include <stdio.h>
int main() {
printf("hello, world");
}
```

save this file as `hello.c`.

There is a bit of a learning curve for each of these editors, but they can be quite useful. Most users will choose `vi` or `nano`. The "standard editor" `ed` is found on almost every machine and can be a good fallback if you need to repair a system or make a quick edit.

C programming side quest

Use the c compiler to create an executable program from our `hello.c` code:

```
furst@snaut:~$ cc hello.c
furst@snaut:~/foobar$ ls
a.out*  hello.c
```

Now we have an executable file `a.out`. Run it by typing `./a.out`. [2]

5 Controlling the terminal output

Several control characters are used to control the terminal and its output:

- `^c` – stop execution (halt a program or clear a terminal line)
- `^s` – pause output

[1] Some people have strong feelings about editors. See https://en.wikipedia.org/wiki/Editor_war.
[2] A right of passage!

- `^q` – continue output
- `^d` – end of transmission / end of file

Here, the carat character `^` stands for `control`. You might use control-c the most.

6 Suspending and backgrounding processes

Sometimes you need to pause what you're doing (like editing a file) to perform another task. You can have multiple terminal windows (or tabs) open, but if you're working on a remote machine, it may be inconvenient to open multiple sessions. Suspending (pausing) execution is one option. In other cases, a program may take a while to run, and its output is written to files, not the terminal. In that case, running it in the background is a good option.

- `^z` – suspend an active process
- `fg` – foreground a suspended or backgrounded process
- `jobs` – display status of jobs in the current session (not all shells)
- `&` – used after a command, this runs a process in the background

Type `cat` and `return`. Then type `^z`.

```
furst@snaut:~$ cat
^Z
[1]+  Stopped                 cat
```

The `cat` command was reading from the standard input (the keyboard). When we typed control-z, it suspended the process. Type `ps` and you should still see it listed as an active process. We can reactivate it using the command `fg`:

```
furst@snaut:~$ fg
cat
```

It tells us that the `cat` command is active again. (Nothing much will happen. Try typing a few lines. What do you see and why?)

Control-z is useful when you are editing a file and need to return to the command line (although some editors have the ability to open a new shell). Use `vi` or `nano` to open your `hello.c` file. Then hit `control-z`:

```
furst@snaut:~/foobar$ vi hello.c

[1]+  Stopped                 vi hello.c
furst@snaut:~/foobar$
```

Now do some other work. Compile the program:

```
furst@anisotropic:~/foobar$ cc hello.c
```

You should see a new file called `a.out`:

```
furst@anisotropic:~/foobar$ ls -l
total 20
-rwxrwxr-x 1 furst furst 15960 Mar 31 21:41 a.out*
-rw-rw-r-- 1 furst furst    65 Mar 31 21:41 hello.c
```

This is the executable or binary file compiled from our short c program. Run it by typing `./a.out`.

Now return to the editor. Type:

```
furst@anisotropic:~/foobar$ fg
vi hello.c
```

(This will put you back in the editor. You probably won't see these lines until you finally quit the editor, unless you're using `ed`.)

> **Warning!** Any process running in the background will be terminated if you close the terminal session. You can keep a process running by using `screen` or `nohup`.

7 More on files: wildcards and matching

In Part I, we used `mv`, `cp`, and `rm`, to manipulate files. These commands accepted the filename. We can select more than one file to act on by using wildcards:

- `*` – match a string of characters
- `?` – match one character

For example, compare the output for the command

```
ef1j@snaut:~$ ls /etc/dev
```

with

```
ef1j@snaut:~$ ls /etc/dev tty*
```

What files are listed in the second example? Now try:

```
ef1j@snaut:~$ ls /etc/dev tty?
```

What is the difference?

Exercise 4: Practice listing certain files. Can you list all of the `/dev/tty` files that begin with `tty1`? How about all of the programs in `/usr/bin` that begin with the letter *p*? Try some other letters!

Exercise 5: Count the number of files that begin with the letter p in `/usr/bin` by typing `ls /usr/bin/p* | wc -l`. What is the program `wc`? (RTFM!)

Delete all files, recursively

Clear out a directory structure for deletion using the command

```
ef1j@snaut:~$ rm -r *
```

Be careful! Remember, there is no undo!

8 Making a backup of a directory side quest

What if I have a directory `~/foobar` with important files? I want to make a copy of that directory. Can I use the following command?

```
ef1j@snaut:~$ cp foobar foobar_backup
```

Why or why not? Try it!

All right, the copy command will not act on a directory. However, we can copy all of the directory contents to a new directory using the recursion option, `cp -r`:

```
ef1j@snaut:~$ cp -r foobar foobar_backup
```

Now we should have a backup of `foobar` with all of the files (and directories). Here's the original directory in my case:

```
furst@snaut:~$ ls -l foobar
total 20
-rwxrwxr-x 1 furst furst 15960 Mar 31 13:52 a.out*
-rw-rw-r-- 1 furst furst    61 Mar 31 13:52 hello.c
```

and here is the backup:

```
furst@snaut:~$ ls -l foobar_backup/
total 20
-rwxrwxr-x 1 furst furst 15960 Mar 31 18:07 a.out*
-rw-rw-r-- 1 furst furst    61 Mar 31 18:07 hello.c
```

Something interesting happened: when we copy the files, they have new modification times. This might be undesirable for a backup. But I can *preserve* the old modification times with the `-p` option:

```
ef1j@snaut:~$ cp -rp foobar foobar_backup
```

Now the copies of those files should preserve their original modification times. This holds for a number of other file transfer and copying commands, like `rsync` and `sftp`.

9 Shell history

Typing in a command again and again can be a drag. Luckily, most shells save a history of previous commands. You can refer back to this history and even execute previous commands.

- `history` – print the shell command history
- `!` and `!!` – execute a previous command or the most recent command

Type `history` (or maybe `history | less`):

```
ef1j@snaut:~$ history
...
 2013  ls
 2014  less weather.sh
 2015  man whoami
 2016  whoami
 2017  man history
 2018  history
```

Now, if I type `!less` ("bang less"),

```
ef1j@snaut:~$ !less
```

it will execute the last `less` instruction in my command history (in this case, `less weather.sh`). This is useful when commands are long and have a lot of options or if you need to repeatedly refer back to a text file, like the example here.

Typing `!!` ("bang-bang") will execute the last command in the history. This is also handy!

10 Shell programming

The shell is the program that is managing our input and output in the terminal. There are several shell programs to choose from including the Bourne shell `sh`, C shell `csh`, korn shell `ksh`, Z shell `zsh`, but `bash` is a common default.

We can also write scripts in the shell. These may be used to run other programs or a combination of tasks. Programming the shell is a subject in itself, but a powerful tool. Here's one (perhaps useful) example to play with.

Type in the following and save it as `weather.sh`:

```
#!/bin/bash
# weather.sh [-s STATE] [ZONE]
# city weather by zone with state option
# Version 2025 APR 01
#
# zones at https://www.weather.gov/pimar/PubZone
```

```
usage() { echo "Usage: $0 [-s STATE] [ZONE]" 1>&2; exit 1; }

# defaults
STATE="de"
ZONE="001"

while getopts "s:" flag
do
    case "${flag}" in
        s) STATE=${OPTARG};;
        *) usage;;
    esac
done
shift $((OPTIND -1))

# more than one option, so do not run
if [ "$#" -ge 2 ]; then
        usage
    else
    if [ "$#" -eq 1 ]; then
        # Use a regex to check if the argument is exactly three zone
           digits
        if [[ ! "$1" =~ ^[0-9]{3}$ ]]; then
            usage
        else
            ZONE=$@
        fi
    fi
fi

# curl command to get the weather forecast
curl -s "https://tgftp.nws.noaa.gov/data/forecasts/zone/${STATE}/${STATE}
   z${ZONE}.txt"
```

Now change the permissions to make this script executable:

```
ef1j@snaut:~$ chmod u+x weather.sh
```

You now have a short script that downloads the latest weather forecast![3] Who needs the web!

The shell script above shows how you can write your own Unix utility. It processes command options like the programs we've learned about and provides some user information, including fairly reasonable error processing.

Exercise 6: Try a few different states and zones. (You'll have to look them up). Redirect each report to a unique file. Can you concatenate these into one large file?

[3] macOS Easter egg. Type: `./weather.sh | tail -n +15 | say`