Slide 1

Administrivia

- Reminder: Homework 1 due next Monday.

Slide 2

Minute Essay From Last Lecture

- Some people had made changes to their `.bashrc` files, some for specific courses (no surprise).
- A people had made additions on their own. Interesting how in this course people come in with very different backgrounds, some knowing quite a bit more than others. "It's all good"?
- Some weren't quite sure! To find out, compare to `/etc/skel/.bashrc`.

Slide 3

Pipes and Filters, Recap/Revisited

- Pipes allow you to connect output of one program to input of another. (There are also "named pipes" that work similarly and are persistent as opposed to single-use.)
- They're particularly attractive when combined with "filter" programs — and UNIX has lots of them, some of which seem kind of silly except for how well they work as building blocks.

Slide 4

Some Filters

- `head`, `tail` get first or last N lines.
- `sort` sorts, `uniq` discards (consecutive) duplicates.
- `grep` searches for text (or regular expression — more later).
  (Name is from very old editor, where `g/re/p` meant "globally search for regular expression and print".)
- `wc` counts characters, words, lines.
- `tr` "translates". Good for converting, e.g., upper-case to lower-case.
- `cat` "concatenates" one or more inputs to output.
- `tee` duplicates input. Good for capturing output to a file while also displaying it onscreen.

Slide 5

Examples

- Find all processes that belong to your username:

```
ps aux | grep $USER
```

- Count lines in all C source files in current directory:

```
cat *.c | wc -l
```

- Show how much space each subdirectory of your home directory is using, sorted by size.

```
du -sk $HOME/* | sort -n
```

(Unfortunately this omits directories starting with a dot.)

Slide 6

More Filters — `sed`

- `sed` ("stream editor") is a non-interactive editor. By default does *not* edit in place, but works as a filter, transforming input to produce output. Especially useful with regular expressions (later), and in manipulating variables within a command (later).
- Some simple uses on next slide, with command inline. For more complicated edits, can put command(s) in a file.

Slide 7

Simple Examples of `sed`

- Search and replace:

```
sed 's/old/new/g' infile > outfile
```

- Delete lines containing some string:

```
sed '/this/d' infile > outfile
```

(How else could you do this?) (`grep -v`!)

Slide 8

More Filters — `awk`

- `awk` is an implementation of programming language AWK ("pattern scanning and processing language", (named after its inventors — as mentioned in its `man` page).
- Lines of AWK program specify pattern and action. (Can also include function definitions.)
- Basic processing: Split each line of input ("record") into "fields", compare to patterns in program, execute actions for any patterns that match.
- Some simple uses on next slide, with command inline. As with `sed`, for more complicated edits, can put command(s) in a file.

Slide 9

Examples of `awk`

- Print selected lines of input:

  ```
  awk '/this/' infile
  ```

  (How else could you do this?) (`grep`)

- Find all users who are running processes on the local machine:

  ```
  ps aux | awk '{ print $1 }' | sort | uniq
  ```

- Generate a list of machines that are "up":

  ```
  ruptime | grep up | awk '{print $1}'
  ```

  (Unfortunately this omits some machines, such as the dias cluster — different subnetwork.)

Slide 10

Still More Filters, and Other Useful Commands

- `diff` compares files or directories. (Useful in "regression testing" of programs, together with I/O redirection.)
- `xargs` "builds and execute command lines from standard input". My standard(?) silly(?) example of the power of the command line:

  ```
  ps aux | grep $USER | awk '{print $2}' | xargs kill
  ```

Slide 11

Still More Useful Commands — `find`

- Very powerful/flexible, though there are so many options you probably won't remember anywhere near all of them. `man` page is useful if daunting!

  Simple examples:

- Find all files in the current directory and subdirectories modified in the last week.

  ```
  find . -mtime -7
  ```

- Find all files in your home directory and subdirectories whose name contains `hello`.

  ```
  find $HOME -name "*hello*"
  ```

  (Double quotes are needed so shell doesn't try to expand wildcard.)

Slide 12

`find`, A Bit More

- Summarizing and simplifying a bit from the `man` page, arguments to `find` consist of paths, "options", "tests", "actions", and "operators".
- Path(s) come first — where you want to search.
- "options" are next and apply to whole command, e.g. `-maxdepth`.
- Then there are "tests" (search criteria), "actions" (what you want to do with files that match — default is to print name), and "operators" (such as logical and, or) connecting them.

  Examples on next slides . . .

Slide 13

Examples of `find`

- Find all files in the current directory and subdirectories that end in `.bak` and remove them.

  ```
  find . -name "*.bak" -exec rm {} \;
  ```

  Here, `-name` is a "test" and `exec` an "action".

- As above, but prompt before executing each `rm`:

  ```
  find . -name "*.bak" -ok rm {} \;
  ```

  Here the "action" is `-ok`. (Might seem like you should be able to just use `rm -i`, but that doesn't work.)

Slide 14

More Examples of `find`

- Find files modified in last 24 hours and sort by modification time:

  ```
  find . -mtime -1 -type f | xargs ls -lt
  ```

  Here there are two "tests" (for time and type) and the default "action" (print), and we pipe into `xargs`

- But the above also lists files in hidden directories `.cache` and `.mozilla`, which we may not care about. To exclude them . . .

Slide 15

More Examples of `find`, continued

- . . . we could type

```
find . -name .cache -prune
     -o -name .mozilla -prune
     -o -mtime -1 -type f | xargs ls -ltd
```

(all on one line)

This has three test-plus-action clauses, connected by `-o` (logical or) — two to tell `find` not to descend into directories we don't want, plus one that selects files we want.

(I use `ls -ltd` because the two "prune" clauses print the names of the pruned directories, and without `-d` `ls` would print their contents.)

Slide 16

Shell Scripts

- What you type as input to a shell is a programming language, and a "shell script" is just a program in this language.
- Normally, first line of script is `#!` ("hash bang") followed by path for shell (`/bin/bash`, e.g.), and the file is marked "executable" (with `chmod`). But you can also execute commands in file `anyfile` via `sh anyfile` (or `bash anyfile`).
- With the exception of the first line, lines starting with `#` are comments.
- (`hello` example.)

Slide 17

Shell Variables

- Define/assign variables with, e.g., `myvar="hello"`. (Note absence of spaces.)
- Reference with, e.g., `$myvar`.
- What's the difference between these and "environment variables" already mentioned? Shell variables are local to the shell, not passed on to child processes. Distinction is somewhat blurred in Bourne shells. Convention is that environment variable names are all caps.

Slide 18

Other Features

- What you type is a programming language, so in addition to variables it has functions, condiional execution, and loops. More next time!

Slide 19

Minute Essay

- What command line could you use to count the number of aliases in your `.bashrc` file?

Slide 20

Minute Essay Answer

- One possible answer:

  ```
  grep alias .bashrc | wc -l
  ```

  (You could add `-w` to `grep` — see `man` page for what that does.)