Slide 1

Administrivia

- None.

Slide 2

Recap — Topics So Far

- UNIX philosophy — lots of little programs that cooperate.
- A little about files — “everything’s a file” (including devices, e.g.); security model.
- A little about processes — one gets started for every command you run; environment variables; foreground and background processes.
- Some stuff about shells — what they do with what you type in; shell customizations (including aliases and functions); I/O redirection.
- Pipes; filter programs.

Slide 3

Filters — Review / A Bit More

- `head`, `tail`.
- `sort`, `uniq`.
- `grep` — search for text (or regular expression — more later).
- `wc` — count characters, words, lines.
- `tr` — "translate". Good for converting, e.g., upper-case to lower-case.

Slide 4

Filters, Continued

- `sed` — "stream editor". Example — convert DOS/Windows-style text file (each line ends with `\r\n`) to UNIX-style (each line ends with `\n`).
- `awk` — "pattern scanning and processing language" — many interesting possibilities; simplest is just to break up input into whitespace-delimited fields.

Slide 5

More Useful Commands

- `find`. Very powerful/flexible, though if you don't use it often you probably will have to read the man page to remember syntax. Examples:
  - Find all files in the current directory created in the last week.

    ```
    (find .  -mtime -7)
    ```
  - Find all files in your home directory whose name contains `hello`.

    ```
    (find $HOME -name "*hello*")
    ```
  - Find all files in the current directory that end in `.bak` and apply `rm -i` to them.

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

Slide 6

More Useful Commands, Continued

- `diff`.
- `xargs`. Example:
  - Find all processes for program `xcpustate` and kill them:

    ```
    (ps aux | grep xcpustate | awk '{ print $2 }'
    | xargs kill)
    ```

Slide 7

Shell Input as a Programming Language

- What `bash` understands is in a sense a programming language, with the shell as its interpreter:
  - Variables (untyped).
  - Expressions (arithmetic and logical).
  - Conditionals (if/then/else) and loops.
  - Functions.
- Can be used interactively, or collected into "scripts".
- I will talk about `bash`, but most shells provide similar functionality, just sometimes with different syntax. If you want to write scripts portable to most Unix systems, probably best to stick to `sh` subset of `bash`.

Slide 8

Shell Scripts

- A "shell script" is just a sequence of things you could type at the shell prompt, collected in a (text) file.
- Normally, first line of script is `#!` 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 `bash anyfile`.
- With the exception of the first line, lines starting with `#` are comments.

Slide 9

Shell Variables

- Define/assign variables with, e.g., `myvar="hello"`. (Notice absence of spaces.)
- Reference with, e.g., `$myvar`.

Slide 10

Quoting and Escape Characters

- Normally `bash` breaks input into "words" based on whitespace, expands wildcards, performs variable substitutions (e.g., `$HOME`), and a fair amount of other stuff.
- When that's not what you want:
  - Precede "special" characters with escape character (backslash).
  - Use double quotes to inhibit all of the above except variable substitution.
  - Use single quotes to inhibit all of the above.

Slide 11

Command Substitution

- Can "inline" output of one command as parameters of another using backquotes. Example:

```
vim `find .  -name "*.c"`
```

- The "inlined" command can even be a pipeline. Example:

```
ls -ld `echo $PATH | sed 's/:/ /g'`
```

Slide 12

Shell Functions and Parameters

- Define functions as described last time — `function` followed by name, parentheses, then function definition in curly brackets. Separate/end commands with `;` or newlines.
- Parameters for functions and shell scripts are positional — `$0` for function name, then `$1`, etc. `$*` is a list of all parameters; `$#` is the count of parameters, not including `$0`.
- Call functions or shell scripts by giving name and then parameters, separated by whitespace. (If a parameter should include whitespace, use quoting or escape characters.)

Slide 13

Conditionals and Loops

- Basic syntax for if/then/else:

  `if` command

  `then` list-of-commands

  `else` list-of-commands

  `fi`

  Which branch is taken depends on return code from command after `if` — 0 considered "true", other values "false".

- Basic syntax for while loops:

  `while` command

  `do` list-of-commands

  `done`

  Continues until return code from command after `while` is non-zero.

Slide 14

Conditionals and Loops, Continued

- Basic syntax for `for` loops:

  `for` var `in` list-of-values

  `do` list-of-commands

  `done`

- Other constructs include `case` (like C `switch`), `until`.

Slide 15

Useful Commands for Conditions, Loops, Etc.

- Probably the most common for conditions is `test`. Many options. Example:

```
if [ -z "$1" ]
then echo Usage:  `basename $0` someparameter; exit
fi
```

- For lists/loops, `seq`, wildcards, and command substitution are good. Examples:

```
for n in `seq -w 0 21`
do echo Xena$n
done

for f in `ls $HOME`
do du -sh $HOME/$f
done
```

Slide 16

Arithmetic

- Most basic/portable way probably `expr`. Example: `n=`expr $n + 1``.
- In `bash`, can also use double parentheses. Example: `n=$((n + 1))`.

Slide 17

Reading from Standard Input

- To read from shell's / script's standard input: `read`. Example:

```
echo "Do you really want to do this?  (y/n)"
read ans
if [ ".$ans" = ".y" ] ....
```

Slide 18

"Here" documents

- We talked about redirecting input and output. One more option for input, useful in scripts, is to get it from the script itself — "here" document. Example:

```
#!/bin/sh
mail -s "a subject" bmassing << EOF
hello
I am here
who are you?
is this fun?
EOF
```

Slide 19

A Few More Useful Things

- `pushd / popd` — manipulate stack of directories.
- `getopt` — process command-line options.

Slide 20

Minute Essay

- The command `ping -c 1 Janus00` will test to see if `Janus00` is network-reachable. Write a few lines of `bash` input that would let you "ping" all the `Janus` machines.

Minute Essay Answer

- One possible answer:

```
for n in `seq -w 0 21`
do
    ping -c 1 Janus$n
done
```

Slide 21