Shell (nouns, selected)

- a hard outer covering of an animal, as the hard case of a mollusk, or either half of the case of a bivalve mollusk
- any of various objects resembling such a covering, as in shape or in being more or less concave or hollow
- the hard exterior of an egg
- a hard, protecting or enclosing case or cover
- an attitude or manner of reserve that usually conceals one's emotions, thoughts, etc.
- a hollow projectile for a cannon, mortar, etc., filled with an explosive charge designed to explode during flight, upon impact, or after penetration
- small pieces of pasta having the shape of a shell
- the lower pastry crust of a pie, tart, or the like, baked before the filling is added

Dictionary.com, "shell," in *Dictionary.com Unabridged*. Source location: Random House, Inc. http://dictionary.reference.com/browse/shell. Available: http://dictionary.reference.com. Accessed: October 04, 2009.

David Notkin ● Autumn 2009 ● CSE303 Lecture 3

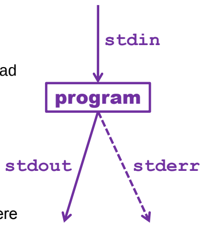
Today

- Combining commands
  - input/output redirection
  - pipes
- Processes and basic process management

I/O streams: standard

- Three I/O streams: **`stdin`**, **`stdout`**, **`stderr`**
- The program itself has statements that read or write to these streams

-
  ```
  #include <stdio.h>
  main() {
     printf("Hello 303\n");
  }
  ```
- **`printf`** is defined to write to **`stdout`**
- So the program doesn't know or care where it is writing output
- Similarly, for reading input or writing errors (using, for example, **`scanf`** and **`fprintf`**)



Output redirection

- This standard allows the shell to provide user-level redirection of I/O
- **`command > filename`**
- Run **`command`** and write its output to **`filename`**
  - That is, hook **`filename`** to **`stdout`** of **`command`** instead of defaulting to the console
  - Take care: existing files are overwritten
- **`>>`** appends to **`filename`** rather than overwriting it
- Again, the program representing **`command`** doesn't manage – or even know anything about – such redirection

Examples

- **`ls -l > myfiles.txt`**
- **`java Foo >> Foo_output.txt`**
- **`cat > somefile.txt`**
  (writes console input to the file until you press **`^D`**)
  - Easy way to create a simple file without using an editor

Miscellaneous

- **`command > /dev/null`** suppresses the output
  - Why might you want to do this?
  - What is **`/dev/null`**?
- Redirecting **`stderr`**: Same idea, with silly syntax (RTFM – Read The &*(@%$% Manual)

- How might you do the equivalent of output redirection in a Windows environment?
- In any GUI-based environment?

Input redirection: same idea for **stdin**

- **`command < filename`**
- Run **`command`** and use **`filename`** as **`stdin`**
  - If the program reads from **`stdin`**, instead of awaiting input from the console, it will instead read the input from a file
- Only works for programs written in terms of **`stdin`** – if a program explicitly reads input from a specific file, that cannot be overridden by the shell
- Remember: arguments/parameters are passed in through the command line, and are unaffected by any redirection

Combine input and output redirection

```
sort -r < /usr/share/dict/linux.words > rev.dict
```

Combining commands

- **`wc /usr/share/dict/linux.words > t`**
- **`grep 0 < t`**

- When the output of one command is used as the input to the next command, there is a lovely shorthand – *pipes* (or sometimes *pipelines*)

- **`wc /usr/share/dict/linux.words | grep 0`**

- This connects the **`stdout`** of **`wc`** to the **`stdin`** of **`grep`**

Examples

```
ls -l | more
grep free /sources/gnu/less/*.c | uniq | sort
grep free /sources/gnu/less/*.c | uniq | sort | wc
grep free /sources/gnu/less/*.c | sort | uniq | wc
grep free /sources/gnu/less/*.c | grep -v freelist
```

Multiple commands

- Less important than pipes, you can also run multiple unrelated commands in the shell
- **`command1 ; command2`**
  - run **`command1`** and then **`command2`** afterward – there is no connection between the programs or their input/output streams
- **`command1 && command2`**
  - run **`command1`**, and if and only if it succeeds, run **`command2`** afterward
- Question: what does it mean for a command to "success" or "fail"?

An unfair, but interesting, comparison

- "Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency."
  –Jon Bentley, Programming Pearls ~1986
- Donald Knuth solution
  - *CACM*, Programming Pearls, June 1986 (Bentley with Knuth and McIlroy)
  - Literate Programming
  - Key data structure: trie
  - Roughly eight pages, including documentation, index, etc.

McIlroy's quotations

- "I found Don Knuth's program convincing as a demonstration of [literate programming] and fascinating for its data structure, but I disagree with it on engineering grounds."
- "A first engineering question to ask is: how often is one likely to have to do this exact task'? Not at all often, I contend. It is plausible, though, that similar, but not identical, problems might arise. A wise engineering solution would produce – or better, exploit – reusable parts."
- "The following shell script was written on the spot and worked on the first try."

McIlroy's solution

```
tr -cs A-Za-z\' '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -k1,1nr -k2 |
sed ${1:-25}q
```

- Make one-word lines by transliterating the complement of the alphabet into newlines and squeezing out multiple newlines.
- Transliterate upper case to lower case.
- Sort to bring identical words together.
- Replace each run of duplicate words with a single representative and include a count
- Sort in reverse numeric order.
- Pass through a stream editor; quit after printing the number of lines designated by the script's first parameter (default is 25)

No, I don't expect you to be able to do this! It's to show some of the power.

Common misuses: pipes and `cat`

- bad: `cat filename | command`
- good: `command < filename`

- bad: `cat filename | more`
- good: `more filename`

- bad: `command | cat`
- good: `command`

Processes

- A set of Unix commands deal with processes – examples include `ps`, `fg`, `bg`, `kill`, …
- What is a process?
- Is it the same as a program? Actually, what is a program?
  - `hello.c`, `hello.s`, `a.out`, …

Rough idea: process

- A process is a running execution of a program
  - Lots of details about processes vary across operating systems – beyond the scope of 303
- When you execute a command, a process is created, the program is instantiated and executed – when the program completes, the process is killed
- If you execute one command twice simultaneously – how would you do this? – then each execution takes place in its own process
  - Each has its own variables, own `stdin/stdout`, can take different branches, doesn't know about the other, etc.

Processes: a bit more

- The operating system has its own processes, too
  - Some manage disks, other manage processes, …
  - In Unix, OS processes are owned by `root` and each process has a unique ID (PID)
- And other users sharing the same operating system have their own processes
- The OS makes sure that each process gets its chance to execute on the CPU(s) – this is called scheduling

Process commands

command	description
`ps`	list processes being run by a user; each process has a unique integer id (PID)
`top`	show which processes are using CPU/memory; also shows stats about the computer *Keeps executing until killed!*
`kill`	terminate a process by PID
`killall`	terminate processes by name

- use **`kill`** or **`killall`** to stop a runaway process (infinite loop)
- similar to **`^C`** hotkey

Background processes

command	description
`&`	(special character) when placed at the end of a command, runs that command in the background
`^Z`	(hotkey) suspends the currently running process
`fg` `bg`	resumes the currently suspended process in either the foreground or background

- You would like some processes to continue while you are doing other things – maybe your editor, maybe a browser, etc.
- You can do this by running some processes “in the background”, so the shell doesn’t have to wait until those processes finish; ex:

```
$ emacs &
```

- If you forget to use **`&`**, suspend your process with **`^Z`**, then run **`bg`**

Searching and sorting: repeat

command	description
`grep`	search a file for a given string
`sort`	convert an input into a sorted output by lines
`uniq`	strip duplicate lines
`find`	search for files within a given directory
`locate`	search for files on the entire system
`which`	shows the complete path of a command

- **`grep`** is a very powerful search tool; more over time

Keyboard shortcuts: repeat

^***KEY*** means hold `Ctrl` and press ***KEY***

key	description
`Up arrow`	repeat previous commands
`Home/End or ^A/^E`	move to start/end of current line
`"`	quotes surround multi-word arguments and arguments containing special characters
`*`	"wildcard" , matches any files; can be used as a prefix, suffix, or partial name
`Tab`	auto-completes a partially typed file/command name
`^C or ^\`	terminates the currently running process
`^D`	end of input; used when a program is reading input from your keyboard and you are finished typing
`^Z`	suspends (pauses) the currently running process
`^S`	don't use this; hides all output until ^G is pressed

File system: repeat

directory	description
`/`	root directory that contains all others (drives do not have letters in Unix)
`/bin`	programs
`/dev`	hardware devices
`/etc`	system configuration files ▪ `/etc/passwd` stores user info ▪ `/etc/shadow` stores passwords
`/home`	users' home directories
`/media,/mnt,...`	drives and removable disks that have been "mounted" for use on this computer
`/proc`	currently running processes (programs)
`/tmp, /var`	temporary files
`/usr`	user-installed programs

Questions?