CSE 374 Programming Concepts and Tools

Lecture 03 - Processes, I/O Redirection

Discussion

Turn and talk with the person next to you.

Try to name all the Linux commands you've learned so far from memory!

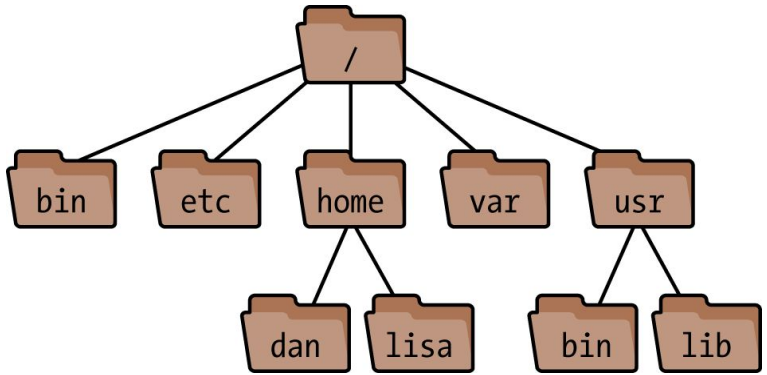
Review

Commands are pre-existing programs

```
program_name [options] [arguments]
```

Think of the file system as a tree

- Absolute vs. Relative paths
- File operations: `ls, cp, mv, rm`
- Directory operations: `cd, pwd, mkdir`
- …



Other Commands: `ssh, scp, wget, tar, echo, clear, …`

Processes, Shell Job Control

Processes in the Shell

Each **program** you run represents one or more **processes**.

- The **program** is the code/instructions
- A **process** is a unit of work on a Linux system.
  - Linux provides commands for viewing and manipulating them.
- Every process is identified by a numeric process ID, or **PID**.
- We can run a given program many times, creating many processes

Commands like `cd` and `ls` are built-in programs

- There are also programs that can be installed (more on this later)

Shell Job Control

A **job** is simply the shell's unit of work.

When you run a command interactively, your current shell tracks it as a job. When the command completes, the associated job disappears.

Only one job can run in the foreground at a time.

- `jobs` List your jobs.
- `&` Run a job in the background. Placed at the end of a command line, the ampersand causes the given command to run as a background job.
  - Example: `vim &`
- `Ctrl-Z` Suspend the current (foreground) job
- `fg` Unsuspend a job, bring it into the foreground
- `bg` Make a suspended job run in the background

Shell Job vs. Process

Processes are part of the operating system, whereas jobs are higher-level constructs known only to the shell in which they’re running.

A running program comprises one or more processes; a job consists of one or more programs executed as a shell command.

```
job > program > process
```

Viewing and Controlling Processes

Figure out what's running:

- **`top`** Monitor resource-intensive processes interactively
- **`ps`** List processes (like Process / Activity Monitor for Windows / MacOS).

Stop processes:

- **`Ctrl-C`** Send kill command immediately. (Shows as `^C`)
- **`kill`** (with options) **`<PID>`**
  - If this doesn't work, add the **`-KILL`** or (equivalently) **`-9`** option: **`kill -9 PID`**

Terminate a shell:

- **`exit`**

Demo: Shell Jobs and Processes

Shell Features

Shell Variables, Search Path,
Aliases, Quoting, Escaping, Math

Shell Variables

Shell variables = string substitution

- Declare variables in the shell to easily refer to a given string
- All variables contain strings
- An undefined variable is empty string

Declare variables in the shell with a name and a string value

```
<var name>="<var string>"
```

- Example: `myvar="myvalue"`

Note: no white space allowed on either side of the "="

Using Shell Variables

Refer to your variable using the "**$**" symbol before the var name

$<var name>

- EX: **echo $myvar**
  - myvalue

Some variables are standard and commonly defined by your shell upon login:

- **HOME**: Your home directory, such as /home/chloe
- **SHELL**: The path to your shell (e.g., /bin/bash)
- **USER**: Your login name
- **PATH**: Your shell search path: directories separated by colons

To list a shell's environment variables, run: **env**

Export Variables

The scope of the variable (i.e., which programs know about it) is, by default, the shell in which it's defined.

To make a variable and its value available to other programs your shell invokes (i.e., subshells), use the export command:

```
export myvar
```

This makes variable available in the initial shell environment. It is now called an **environment variable**.

If a program changes the value of an exported variable, it **does not** change the value **outside of the program**.

Quoting

Double quotes can be used to wrap a string with white space into a single argument; retain all but $, `, \

- Example: `myvar="Some string"`

Single quotes tell the shell to treat the string as a literal

- No variable expansion or command substitution
- Example: `echo '$myvar'`
  - `$myvar`
- Example: `echo "$myvar"`
  - `Some string`

Escaping

If a character has special meaning to the shell but you want it used literally (e.g., `$` as a literal dollar sign rather than a variable to expand), precede the character with the backward slash "`\`" character.

- This is called *escaping* the special character.

```
echo "I live in $HOME"
```
(Print a variable value)

```
I live in /Users/chloe
```

```
echo "I live in \$HOME"
```
(A literal dollar sign)

```
I live in $HOME
```

Backquotes

Backquotes (“backticks”) cause their contents to be evaluated as a shell command.

- Example:

```
current_date=`date`
echo "Current date and time: $current_date"
```

Will output:

```
Current Date: Fri Jun 27 10:50:00 AM PDT 2025
```

A dollar sign and parentheses are equivalent to backquotes:

```
current_date=$(date)
echo "Current date and time: $current_date"
```

Math

Everything is interpreted as a string, no concept of integer values

To do math use double parentheses to interpret as arithmetic expression

- **`RES=$((1+2))`**
  - `$RES` will be set to “3”
- **`MYVAR=$((RES+2))`**
  - `$MYVAR` will be “5”
- `k=$i + $j` does not add numbers
  - **`k=$(($i+$j))`** will add numbers
  - Equivalently: `let k="$i+$j"`

Polling Time!

Please answer in the LP3 assignment on Gradescope!

What individual arguments are passed to the following `echo` command?

```
$ echo the "quick brown" "" 'fox "jumps" over'
```

the	quick brown	fox	jumps	over
the	quick brown	*<empty>*	fox "jumps" over	
the	quick brown	*<empty>*	fox jumps over	
the	"quick brown"	""	'fox "jumps" over'	

Search Path

Programs are scattered all over the Linux filesystem, in directories like /bin and /usr/bin. When you run a program via a shell command, how does the shell find it?

The critical variable **PATH** tells the shell where to look.

- Whenever a command is executed there is a list of predefined locations bash looks
- PATH holds the list of pre-designed directories
- **`echo $PATH`**

To add directories to your shell's search path **temporarily**, modify its PATH variable. For example, to append `/usr/sbin` to your shell's search path:

`PATH=$PATH:/usr/sbin`

Configuration Files

To make PATH modifications permanent, modify the PATH variable in your startup file ~/.bash_profile.

.bashrc file

- A shell script that is automatically run whenever a new Bash shell is opened
- Used to initialize your shell state
- Use for commands that are re-run
  - Aliases & functions
- The **rc** suffix stands for “run commands” and is seen in other configuration files.
  - “Commands that bash should run”

.bash_profile file: Use for commands that run once (runs when you login to

which

The **which** command shows *which* absolute path is associated with a given program.

This is very useful for debugging programs you've installed.

If nothing is returned from *which,* your shell will not be able to execute the command unless you specify the command's full file path.

For example,

- `which cd    => /usr/bin/cd`
- `which which => /usr/bin/which`

Alias

Defines a shortcut or “alias” to a command

Rename a bash command, create your own shortcut

`alias <string>=“substitution string”`

- EX: `alias cheer=“echo hip hip hooray!”`
  - Will output “hip hip hooray” on screen when you run `cheer`
- Technically just string replacement
- Only exists within the current state of your shell
- Can set an alias in your **`.bashrc`** file to preserve the alias across all shells

Source

The **source** command evaluates and runs the commands in a file in the current shell, and alters the environment.

This is often used to add environment variables, functions, and aliases to your current session.

- Ex: If I make a change to my **.aliases**, I want them to be applied in my current terminal without re-opening a new one.

```
source ~/.aliases
```

Demo: PATH, Aliases, Export

Input & Output

Input, Output, I/O Redirections,
Pipes, Combining Commands

Review: Input & Output

Inputs come in two main forms: arguments and stdin

Arguments are a list of strings separated by spaces given after the command

- EX: `wc file1 file2`
  - Arguments: “`file1`” and “`file2`”
- Arguments with spaces need to be wrapped in quotes
  - EX: `echo “hello world”`

stdin or Standard Input is a **stream** that the user enters into the terminal

Outputs go to stdout and stderr (for errors)

- Outputs can be redirected to an output file

Input / Output Streams

stdin – Standard in (File Descriptor = **0**)

- Keyboard input typed into the terminal

stdout – Standard out (File Descriptor = **1**)

- Results of a process after it has completed printed to the screen of the terminal

stderr – Standard error (File Descriptor = **2**)

- Results of a process if it exits in error
- By default, the stderr stream is not displayed on the screen

File descriptors are specific to the context of a running process

Input/Output Redirection

The shell can **redirect** standard input, standard output, and standard error to and from **files**.

In other words, any command that reads from standard input can have its input come from a file instead with the shell's **<** operator:

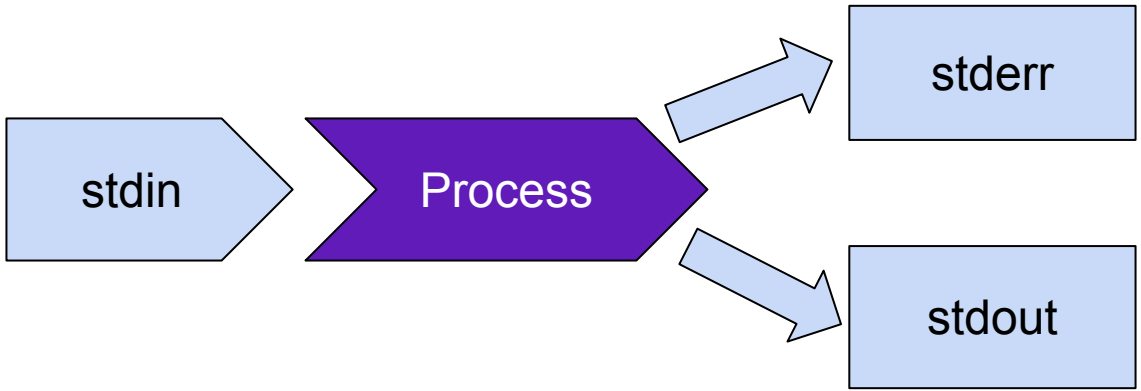
```
command < infile
```

Likewise, any command that writes to standard output can write to a file instead:

```
command > outfile
```
Create/overwrite outfile

```
command >> outfile
```
Append to outfile

Processes all can take INPUT from one source, the default being stdin.

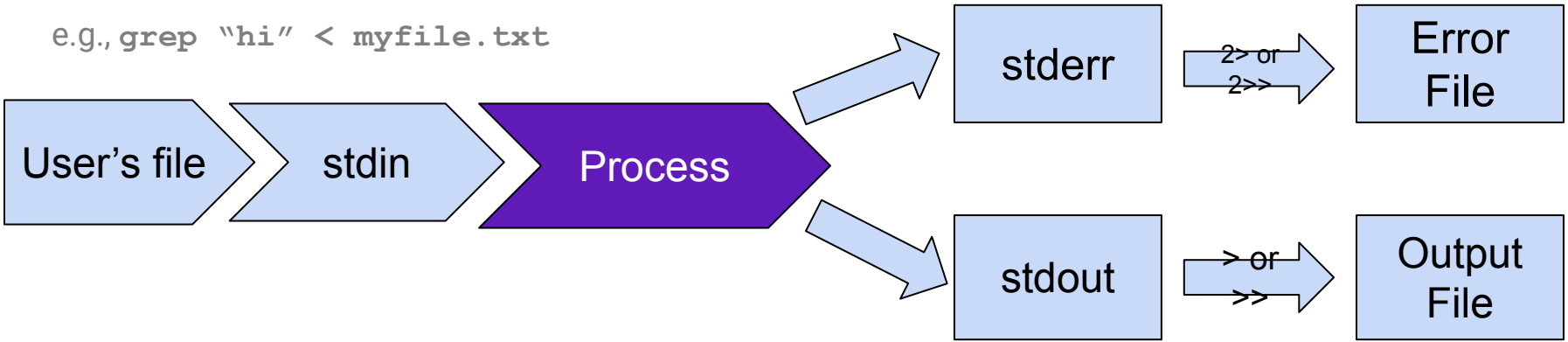


Processes have two OUTPUT destinations, the default being stdout and stderr. You can think of these as two potential files to which a processes can write.

But, instead of using stdin you can use any file, and 'redirect' it in to a process's stdin by using the '<' symbol (pointing towards process).

e.g., `grep "hi" < myfile.txt`

You can also write to different files instead of stderr or stdout.



Examples:

```
grep "hi" myfile.txt >> instances_of_hi
cat nonexistent_file 2> cat_errors_file
```

Redirection Syntax

redirect stdout to a file	`command > file`
redirect stderr to a file	`command 2> file`
redirect stdout to stderr	`command 1>&2`
redirect stderr to stdout	`command 2>&1`
redirect stderr and stdout to a file	`command &> file`

Polling Time!

Please answer in the LP3 assignment on Gradescope!

What does this command do: `foo < bar`

- Prints a boolean representing whether `foo` is less than `bar`.
- Runs the program `bar`, saving stdout to file `foo`.
- Runs the program `foo`, with stdin from file `bar`.
- Runs the program `bar`, saving stdout to shell variable `foo`.

Pipes

We can feed the stdout of one process to the stdin of another using a pipe (“|”)

- Data flows from process to the other through multiple transformations seamlessly
- Similar to redirection, but specifically passes *streams* into other programs instead of their defaults

For example: **`cat file.txt | sort`**

- Sends the output of `cat` into the `sort` program, printing an alphabetically sorted list of line contents.

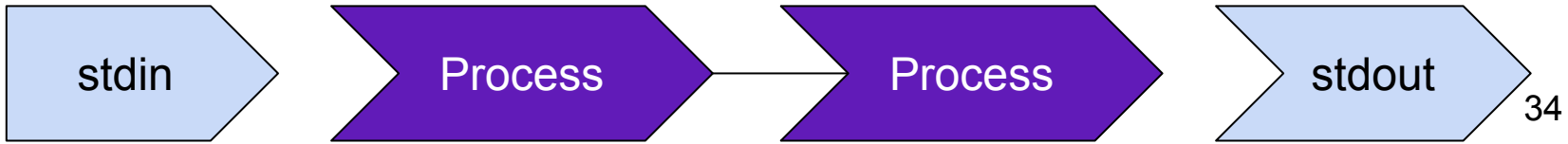
Piping is effective when you have one set of data that needs to be transformed multiple times. Multiple pipes work too.

What is the difference between | and >?

Pipe is used to pass output to another program or utility

Redirect is used to pass output to either a file or stream

- **`thing1 > thing2`** runs `thing1` and then sends the stdout stream to `thing2`, if these are files `thing2` will be overwritten
- **`thing1 > tempFile && thing2 < tempFile`** sends stdout of `thing1` to stdin of `thing2` without overwriting files
  - Equivalent to **`thing1 | thing2`** ; pipes are much more elegant!



Combining commands

To invoke several commands in sequence on a single command line, separate them with semicolons:

```
command1 ; command2 ; command3
```

To run a sequence of commands as before, but stop execution if any of them fails, separate them with && (“and”) symbols:

```
command1 && command2 && command3
```

To run a sequence of commands, stopping execution as soon as one succeeds, separate them with || (“or”) symbols:

```
command1 || command2 || command3
```

Unix Philosophy

A set of cultural norms and philosophical approaches to minimalist, **modular** software development (Wiki).

- Write programs that do one thing well.
- Write programs to work together (I/O).

`ls | wc -l` counts the number of files in the current directory

Linux and MacOS are Unix-based systems.

This philosophy is very important to quality software engineering.

Demo: I/O Redirections

Assignment Reminders

EX0: Course Policies due **tonight @11:59pm**
EX2 due Monday 6/30 **@10:49am**

HW0: Shell Access due **tonight** @11:59pm

HW1: Bash Commands will be released after class and is due next Wednesday 7/2 @11:59

- Become more familiar with running commands
- Relatively short assignment; try to submit on time and avoid using late days

Pre-Course Survey due on Sunday 6/29 @11:59pm

- We (the staff) want to get to know you better and how we can best support you! No late submissions!

Make sure to submit LP3 before the next class as well.

All of these assignments can be found on our course's Gradescope page.

Extra notes

Bonus Slides!

Package Managers

What if built-in features aren't enough?

Some capabilities aren't built-in to most shells

- Graphics/Image Processing
- Specialized audio and video encoders
- Different versions of programming languages (e.g., python 2.7 vs. python 3.12)
- Other programming ecosystems: Node.js, Rust, OCaml, etc.

We don't want to have to re-implement each of these ourselves!

- This would violate the modularity principle of Linux/Unix systems

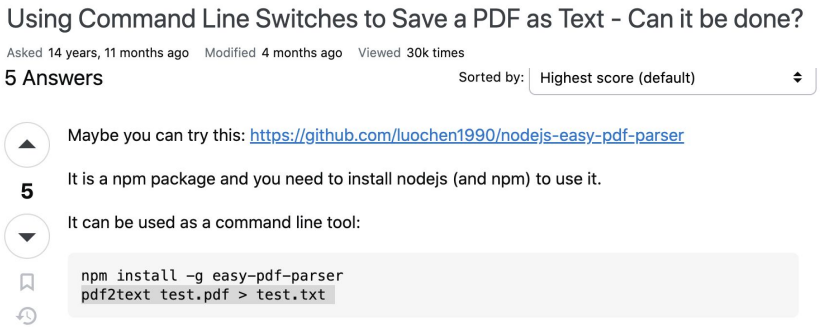
Package Managers to the rescue!

- Allow easy in-shell access to installing and updating various software
- Help manage software requirements (called dependencies)
- Differ from operating system to operating system
  - e.g., `apt-get install` on Debian-based Linux systems
    - You don't get super-user access on Calgary, and thus cannot use apt-get install :/
  - e.g., `brew install` on Mac OS systems (need to install Homebrew first)
- Ensure that installed command line programs are globally available, i.e., on your **`$PATH`**
- Maintain repositories of open-source, free to install software
  - When you want to install a new version of python, for example, it will search these repositories
  - Include OS and architecture-specific versions

Pros and Cons of Using Package Managers

- Free and convenient
- No reinventing the wheel!
- Often included as part of a solution to a problem, install instructions on Github, etc.
- Usually have similar usage patterns, so learning one helps with learning others

- Software may contain bugs
  - e.g., the Heartbleed vulnerability in OpenSSL
  - Use at your own risk!
  - However, most of industry also relies on the same open source software, so… ¯_(ツ)_/¯
- Not always a viable option -- may require super user access
  - Not all package managers require super user access, e.g. Homebrew on Mac



From StackOverflow -- one of the top results from searching “how to turn pdfs into text on the command line”

Extra notes