CSE 374 Lecture 3

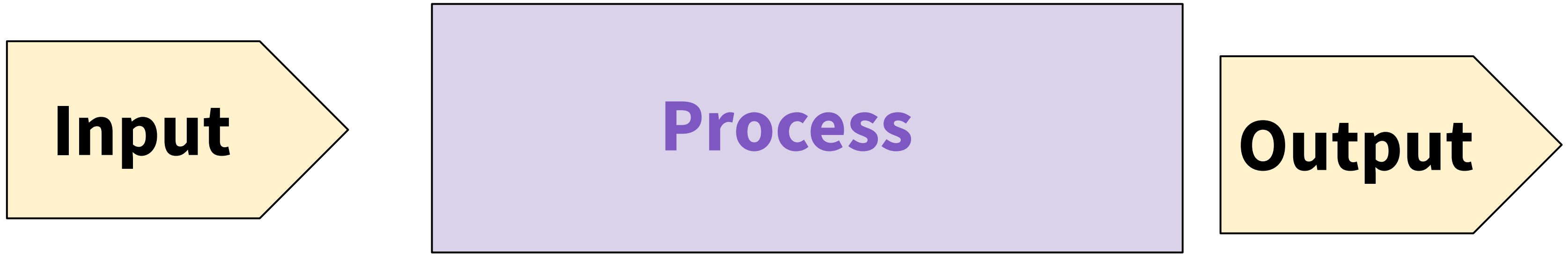
Emacs and I/O Redirection

HW0

- ➔ Due on Monday
- ➔ Goal: ensure that you can use the tools we need for this course
  - ◆ If you have problems, follow up to correct them ASAP.
- ➔ Debugging: in this course you need to work more independently. Use all the resources you have to find answers.
  - ◆ Ex: scp - how could you figure out that command?
- ➔ Note about autograding: Autograding is wonderful for fast feedback, but, can be really challenging to give meaningful feedback & account for all variability.
  - ◆ Not all assignments will be entirely autograded

Bash (shell) Language

- Bash acts as a language interpreter
  - Commands are subroutines with arguments
  - Bash interprets the arguments & calls subroutine
  - Bash also has its own variables and logic



BASH applies its own processing to the I/O text - ‘globbing’

Special Characters

- Directory Shortcuts
  - ~uname or ~
  - ./ or ../
- Wildcards - *Globbing*
  - 0 or more chars: *
  - Exactly 1 char: ?
  - Specified chars: [a-f]

History, or ‘!’

Special Characters

! > < & | * ~ [] “ ‘ ` $ /

What do they all mean?

\ is escape character

“string” → **Would substitute things like $VAR**

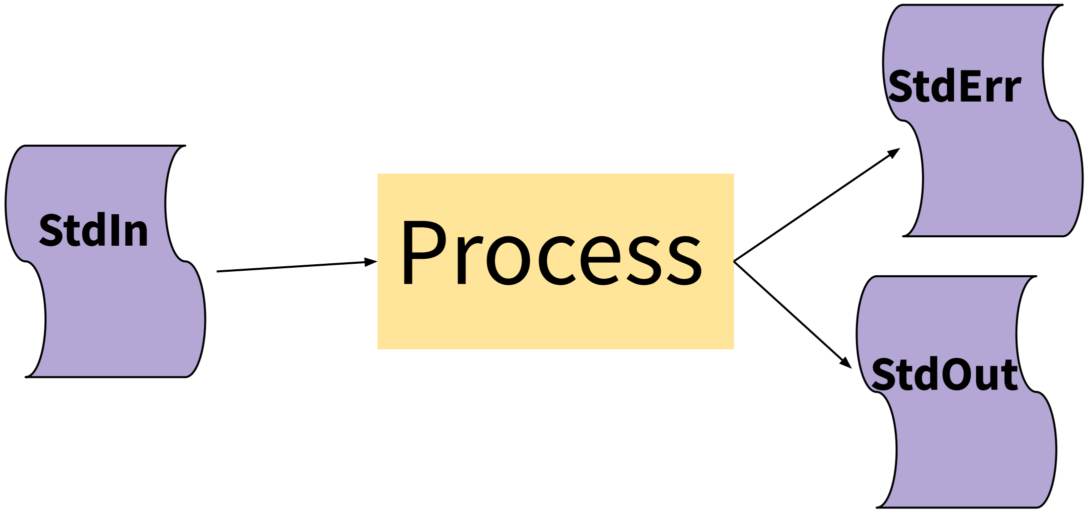
‘string’ → **Suppresses substitutions**

Shell Behavior

All redirection & string expansion or substitutions are done by the shell, before the command.

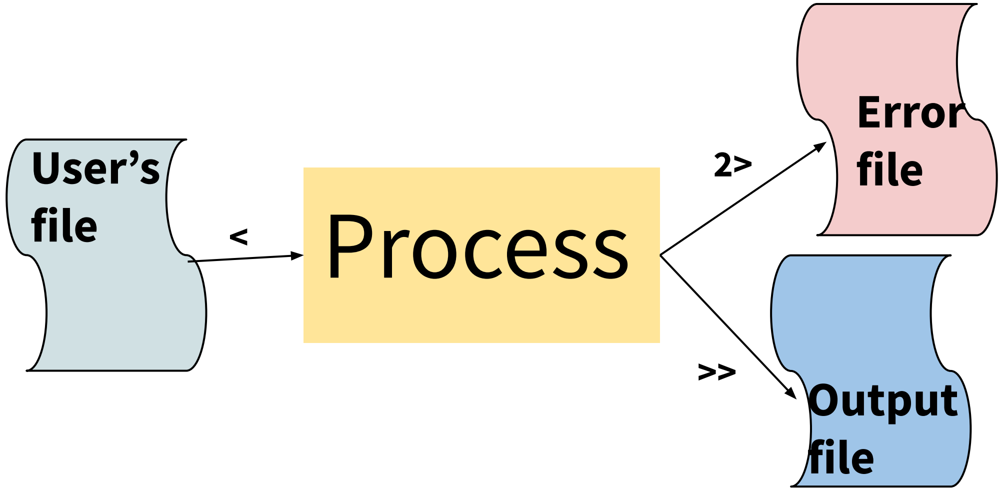
Command only sees resulting I/O streams.

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 by using the '<' symbol (pointing towards process).



You can also write to different files instead of StdErr or StdOut. The '>' symbol means to put in an new file, while '>>' means to append to the end of a file. The '2' specifies that you want iostream '2', or the error stream.

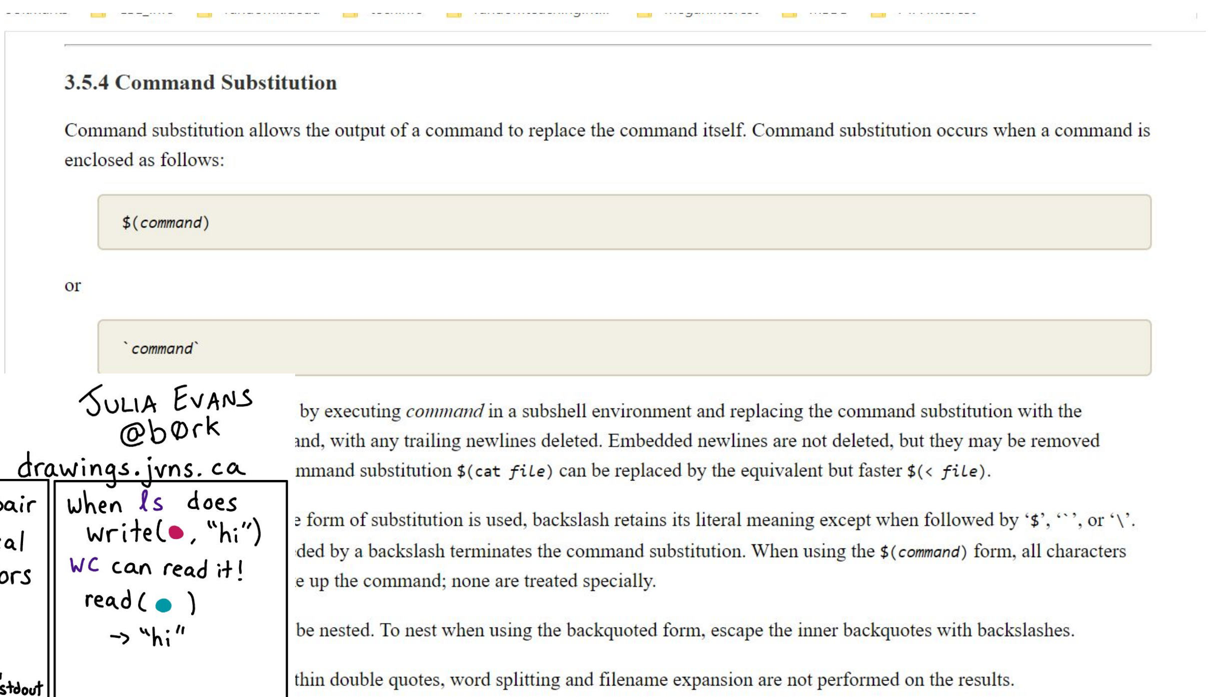
I/O Streams

- All bash commands have three streams
    - 0- stdIn [keyboard]
    - 1- stdOut [screen]
    - 2-stdErr [screen]
- Can redirect streams
    - < yourInput
    - > yourOutput
    - >> appendYourOutput
    - 2> yourError
    - &> yourOutput&Error
    - And more…
- Special File /dev/null
    - Is EoF if input
    - Data is discarded if output
- Can combine one cmd to the next
    - Cmd1 | cmd2 - pipe output of cmd1 into input of cmd2
    - Cmd1; cmd2 - do one after another
    - Cmd1 `cmd2` - use output of cmd2 as argument to cmd1
- Can use cmd logic
    - Cmd1 || cmd2 - do cmd2 if cmd1 fails
    - Cmd1 && cmd2 - do cmd 2 if cmd1 succeeds

Some Bash redirection syntax

redirect stdout to a file →	*command* ***> output***
redirect stderr to a file	*command* ***2> output***
redirect stdout to stderr	*command* ***1>&2 output***
redirect stderr to stdout	*command* ***2>&1 output***
redirect stderr and stdout to a file	*command* ***&> output***

Reading:Bash Redirections (spec), bash hackers redirections (examples)



3.5.4 Command Substitution

Command substitution allows the output of a command to replace the command itself. Command substitution occurs when a command is enclosed as follows:

```
$(command)
```

or

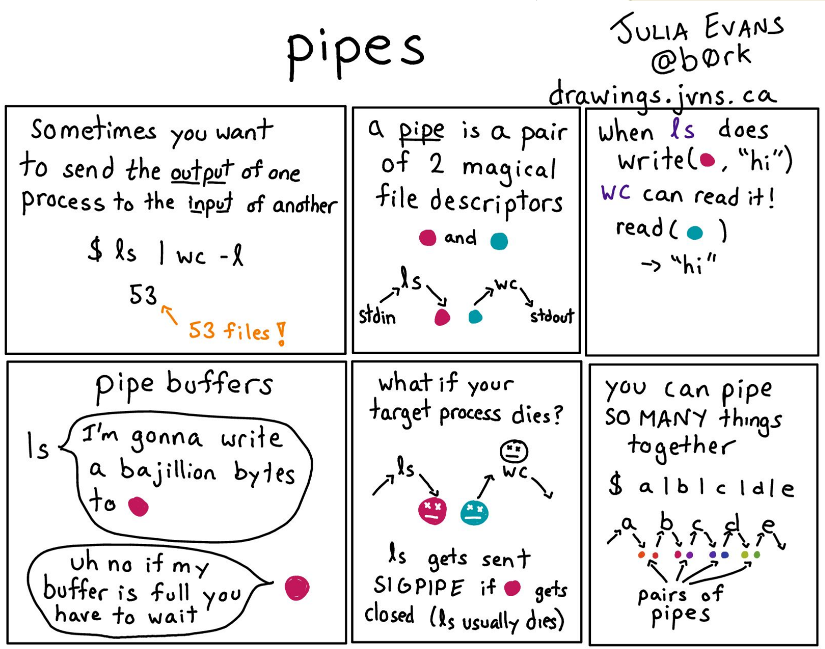
```
`command`
```

by executing *command* in a subshell environment and replacing the command substitution with the
and, with any trailing newlines deleted. Embedded newlines are not deleted, but they may be removed
mmand substitution `$(cat file)` can be replaced by the equivalent but faster `$(< file)`.

e form of substitution is used, backslash retains its literal meaning except when followed by '$', '`', or '\'.
ded by a backslash terminates the command substitution. When using the `$(command)` form, all characters
e up the command; none are treated specially.

be nested. To nest when using the backquoted form, escape the inner backquotes with backslashes.

thin double quotes, word splitting and filename expansion are not performed on the results.


pipes
JULIA EVANS
@b0rk
drawings.jvns.ca
Sometimes you want to send the output of one process to the input of another
$ ls | wc -l
53
53 files!
a pipe is a pair of 2 magical file descriptors
● and ●
ls
wc
stdin
stdout
when ls does write(●, "hi")
wc can read it!
read(●)
-> "hi"
pipe buffers
ls
I'm gonna write a bajillion bytes to ●
uh no if my buffer is full you have to wait
what if your target process dies?
ls
wc
ls gets sent SIGPIPE if ● gets closed (ls usually dies)
you can pipe SO MANY things together
$ a|b|c|d|e
a b c d e
pairs of pipes

Alias

Defines a shortcut or ‘alias’ to a command.

(Essentially a really easy script)

Also, ‘alias’

.bashrc

Variables & Alias

Define variable

i=15

Access variable

$i

Undefined variable is empty string

Alias cheer=”echo yahoo\!”

Towards Scripts

- Shell has a state (working directory, user, aliases, history, streams)
- Can expand state with variables
- ‘Source’ runs a file and changes state
- Can run a file without changing state by running script in new shell.

Emacs (text editor)

```
C-x C-s #save

C-x C-c # quit

C-e # go to end of line

C-a # go to beginning of line

C-x C-f # find a file

C-g #exit menu

C-x C-k # kill a buffer
```

You can use any text editor you like. Emacs is amazingly powerful, and highly customizable with lisp scripts. It is probably worth learning.

Okay, lets make a script!

1. First line of file is #!/bin/bash (specifies which interpreter to execute)
2. Make file executable (chmod u+x)
3. Run a file ./myNewScript
4. Shell sees the shell program (/bin/bash) and launches it to run the script
5. Can include
   a. String tests (string returns true if non-zero length, string < string, etc.)
   b. Logic (&&,||,!) - use double brackets
   c. File tests (-d : is directory, -f: is file, -w: file has write permission etc.)
   d. Math - use double parens

Script Arguments & Errors

Script refers to ith argument at $i ; $0 is the program

Use ‘shift’ to move arguments towards left ($i become $i-n)

Exit your shell with 0 (normal) or 1 (error)

Exit Codes

Command 'exit' exits a shell, and ends a shell-script program.

Exit with no error:

Use `exit` or `exit 0`

Exit with error:

User `exit 1` or.. {1-255}

Variables useful in a script

$# stores number of parameters (strings) entered

$0 first string entered - the command name

$N returns the Nth argument

$? Returns state of last exit

$* returns all the arguments

$@ returns a space separated string with each argument

(* returns one word with spaces, @ returns a list of words)