UNIX streams, pipes, scripts

Jon Chernus (adapted from Ryan Minster)

Department of Human Genetics
School of Public Health
University of Pittsburgh

Document created: September 23, 2024

Location

This slide set is called `unix_streams_pipes_scripts` and is located in the "16_unix_streams_pipes_scripts" folder of our Lectures repository.

Objectives

- To learn how streams operate in Unix
- To learn out to pass streamed data from program to program in Unix
- To learn how to write a script that can run in Unix
- To learn about the cluster and how to submit jobs there

Most of the content in this slide set is essential. You will need to become proficient with it to proceed in the course and to work at the command line in general.

Much of this is a review of what you read in the assigned Active Reading.

Reminder: warnings

- `rm` and `rmdir` are forever
- overwriting is forever (so be careful with `cp`, `mv`, etc.)
- a single whitespace matters a lot
  - `rm -rf a*`
    - recursively delete any file/folder starting with an `a`
  - `rm -rf a *`
    - recursively delete any file/folder starting with an `a`
    - then do the same for *everything* (because `*` matches anything)
- these are all different, so be careful when copy-pasting
  - “,”, and ” (straight vs. curly double-quotes)
  - ‘,’,’, and ‘ (straight vs. curly single-quotes vs. back-tick)
  - -, –, and — (hyphen, en dash, em dash)
- don’t use spaces and special characters in file names
- don’t work on the login node of htc (use `srun -M teach -A hugen2071-2024f --pty bash`)

Reminder: logging on to htc

First, log into the VPN with GlobalProtect.

There are two ways to log on to the cluster

- Via the web: ondemand.htc.crc.pitt.edu
- Via a terminal window
  - `ssh <your_user_name>@htc.crc.pitt.edu`

For details, see
https://crc.pitt.edu/getting-started/accessing-cluster.

Next, always start an interactive job: `srun -M teach -A hugen2071-2024f --pty bash` .

Note

The reading for today covered a few topics that we will address in a later lecture:

- Loops
- `sort` and `uniq`

Streams

Many (not all) of the most useful UNIX commands operate on streams, which you can think of as text data flowing from a source.

- Streams avoid loading an entire dataset into memory
- Standard in: the stream "into" a program
- Standard out: the stream "out of" a program (to the screen by default)
- Standard error: the stream of error messages/warnings from a program (to the screen by default)

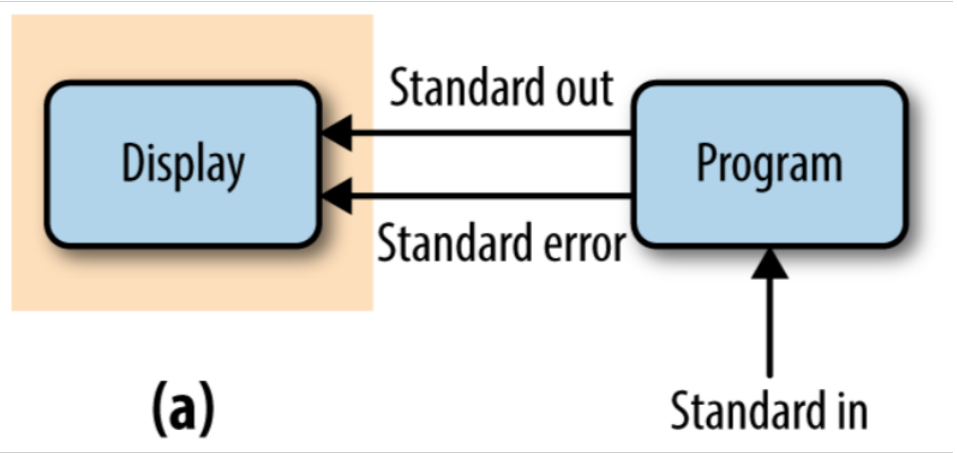

Figure 1: Standard out and standard error print to the screen

Redirection

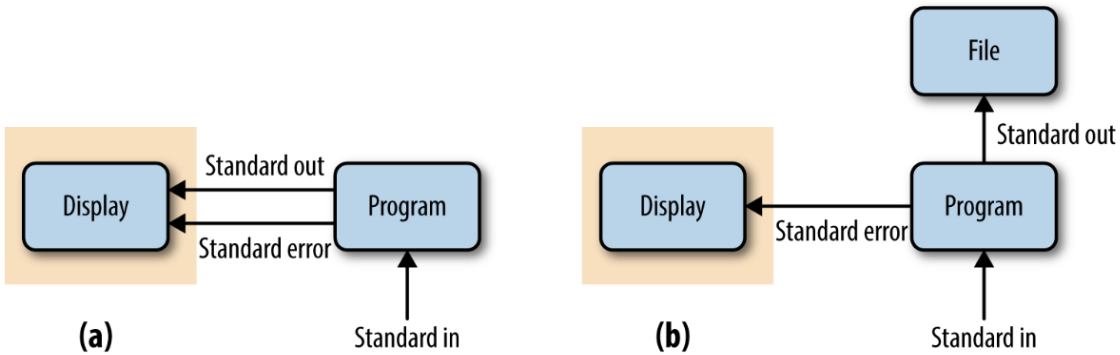

Figure 2: Redirection (of just standard out, to a file)

- Printing everything to the screen is not so useful, so we use **redirection** to make standard out and standard error "go" elsewhere
- You have to give them destinations
  - to a file (which you can either append or overwrite)
  - to `/dev/null` (you can make unwanted output disappear here)
  - they can go to the same or different places

Redirection syntax

Operator	Meaning	Example	What it does
> or >>	Redirect standard out	`command >> results.txt`	Append just stdout to results.txt
2> or 2>>	Redirect standard error	`command > results.txt 2>> log.txt`	Redirect stdout to results.txt and append stderr to log.txt
&> or &>>	Redirect both (to the same file)	`command &> results.txt`	Redirect stdout and std err to results.txt
2>&1	Combine stderr into stdout	`command 2>&1`	*Looks* the same as doing `command`, but stdout and stderr are now in one stream

- Using > overwrites the destination file
- Using >> appends to the destination file

Redirection examples

```
cd data

# The initial files
ls
chr1.txt
chr2.txt

echo "This is my log file" > log.txt

cat chr1.txt chr2.txt chr3.txt > markers.txt 2>> log.txt # Try "merging" together 3 files

# Notice there two more files now
ls
chr1.txt
chr2.txt
log.txt
markers.txt

#Look at markers.txt and log.txt
cat markers.txt # Look at output
chr1:36926582
chr1:66782904
chr1:77840389
chr2:60318540
chr2:85739014


cat log.txt # Look at error log
This is my log file
cat: chr3.txt: No such file or directory

rm markers.txt log.txt # Clear existing output
```

Redirecting standard in

Sometimes it's convenient to redirect stdin to combine streams.

```
# Suppose we want to filter this file and retain the header
cat data2/locations.txt
chr bp
1   36926582
1   66782904
1   77840389
2   60318540
2   85739014

# This will keep only lines starting with 1 but removes the header
grep "^1" data2/locations.txt
1   36926582
1   66782904
1   77840389

# Use < to combine streams and keep the header
cat <(head -n1 data2/locations.txt) <(grep "^1" data2/locations.txt)
head -n1 data2/locations.txt
grep "^1" data2/locations.txt
chr bp
1   36926582
1   66782904
1   77840389
```

Note:

- the order of the commands
- the parentheses
- the lack of spaces in <(

Pipes

What if you want to send output from one program to **another program** instead of to a file?

- **Piping** is like redirection, but to programs instead of files
- stdout of one program becomes stdin of another, and so on
- the syntax looks like:

`command | program1 | program2 | program3.`

- (think of it as analogous to %>% in tidyverse)

Piping example 1

```
# Peek at the first few lines
head -n3 data3/tb1.fasta
>gi|385663969|gb|JQ900508.1| Zea mays subsp. mexicana isolate IS9 teosinte branched 1 (tb1) gene, complete
GCCAGGACCTAGAGAGGGGAGCGTGGAGAGGGCATCAGGGGGCCTTGGAGTCCCATCAGTAAAGCACATG
TTTCCTTTCTGTGATTCCTCAAGCCCCATGGACTTACCGCTTTACCAACAACTGCAGCTAAGCCCGTCTT

# First get non-header lines; then find lines containing non-ACGT letters (regardless of case)
grep -v "^>" data3/tb1.fasta | grep --color -i "[^ACGT]"
CCCCAAAGACGGACCAATCCAGCAGCTTCTACTGCTAYCCATGCTCCCCTCCCTTCGCCGCCGCCGACGC

# Suppose we want to extract lines 10-15
# Use tail to start at line 10, and use head to get the correct number of lines in total
tail -n+10 data3/tb1.fasta | head -n6
ACACGTCCAAGTCCGCCATCCAGGAGATCATGGCCGACGACGCGTCTTCGGAGTGCGTGGAGGACGGCTC
CAGCAGCCTCTCCGTCGACGGCAAGCACAACCCGGCAGAGCAGCTGGGAGGAGGAGGAGATCAGAAGCCC
AAGGGTAATTGCCGCGGCGAGGGGAAGAAGCCGGCCAAGGCAAGTAAGGCGGCGGCCACCCCGAAGCCGC
CAAGAAAATCGGCCAATAACGCACACCAGGTCCCCGACAAGGAGACGAGGGCGAAAGCGAGGGAGAGGGC
GAGGGAGCGGACCAAGGAGAAGCACCGGATGCGCTGGGTAAAGCTTGCTTCAGCAATTGACGTGGAGGCG
GCGGCTGCCTCGGGGCCGAGCGACAGGCCGAGCTCGAACAATTTGAGCCACCACTCATCGTTGTCCATGA
```

Piping example 2

Piping and redirection can be used together

Note how

- \ is used as an escape character to continue the command on a new line
- - means "whatever is coming through the pipe" (like . in tidyverse)

```
cd data4
# toy.* exist; toy1.* do not exist
ls
toy.map
toy.ped

# Notice how the following command continues over several lines!
cat toy.map toy1.map 2> toy.log | \
cat - toy.ped toy1.ped \
2>> toy.log > toy.out

cat toy.out
1   rs0 0   1000
1   rs10    0   1001
1   1000000000  0   0   1   1   0   0   1   1
1   1000000001  0   0   1   2   1   1   1   2
cat toy.log
cat: toy1.map: No such file or directory
cat: toy1.ped: No such file or directory
rm toy.out toy.log
```

`tee` - what if you want to pipe and redirect at the same time?

| sends stdout to another program (invisibly), and > sends it to a file (recording it)

- So what if you want to redirect and pipe at the same time?
- `tee` lets you do both: pipe output into another command *while* also storing it into a file
- Basic syntax: just type `tee filename` after a pipe:
- Pseudo-example: `command1 | tee file.txt | command2`

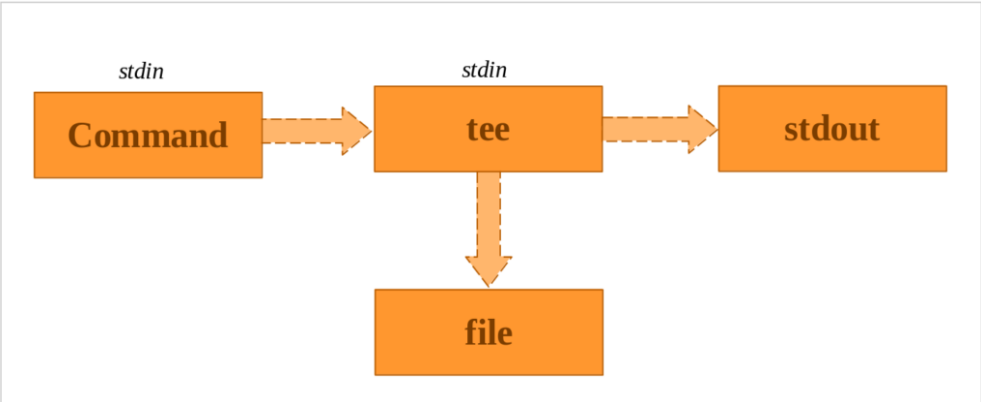

Figure 3: Source: https://www.2daygeek.com/linux-tee-command-examples/

Shell scripts

- A script is a plaintext file usually with extension `.sh`
- Structure
  - required header line (`#!/bin/bash`)
  - should include
    - `set -e` (quit running if there's an error)
    - `set -u` (treat an un-set variable as an error)
    - `set -o pipefail` (prevents masking of errors in a pipeline; more on that later)
    - `set -x` (optional; commands are echoed in the output)
  - then your commands/pipeline
- execute with a command like `bash script.sh` or `./script.sh` (may need to do `chmod u+x script.sh` first)

Shell script example

```
cat scripts/script1.sh
#!/bin/bash
set -euo pipefail

# This line is a comment

echo "Hi there!"

#### This is the last line of the script and does nothing.

bash scripts/script1.sh
Hi there!
```

Variables and command-line arguments

A script is more useful if it can take arguments. You can supply arguments to a bash script like this:

```
bash script.sh argument1 argument2 argument3
```

To use the arguments inside the script, call them by the variables `"$1"`, `"$2"` , `"$3"`, and so on. (Quotes aren't strictly necessary, but are good in case there are spaces in the variable names.) `"$@"` stores all of the command-line variables together.

Assign new variables with = and then use them with $.

Variables example

```
cat scripts/script2.sh
#!/bin/bash
set -euo pipefail

# This script takes two arguments
#    argument 1 is a string to search for
#    argument 2 is a directory
# The script counts how many files in the directory contain the given string in their names

# List the contents, use grep to check for the string, print number of rows of output
ls "$2" | grep "$1" | wc -l

#### This is the last line of the script and does nothing.

bash scripts/script2.sh chr ./data/
       2
```

Running a script with Slurm

- Intensive scripts should be scheduled with a workload manager - our cluster uses Slurm
- To submit your script as a job to the cluster
  - Add `#SBATCH` options immediately below the header (`#!/bin/bash`)
  - Run it with a command like `sbatch script.sh`
  - The job is assigned an ID
- To load programs like R, PLINK, etc. you need to load them as modules
  - enter a command like `module spider r` to learn how to load the module you need
- More info: https://crc.pitt.edu/getting-started/running-jobs-slurm

Slurm scripts in this course

All of your Slurm scripts should start like this:

```
#!/bin/bash
#SBATCH -M teach
#SBATCH -A hugen2071-2024f
```

Example Slurm script

Here's a slurm script:

```
cat scripts/test.sh
#!/bin/bash
#SBATCH --mail-type=BEGIN,END,FAIL
#SBATCH --mail-user=jmc108@pitt.edu
#SBATCH -t 1:00:00

# Have the script quit running if there's an error
set -e; set -u; set -o pipefail

# Print the start time to test.log, then a message, then wait 10 seconds, then the end time
# Print a message (which will go to the slurm log file, since it was not redirected)
echo Job started at `date` > test.log
echo Hello World! >> test.log
echo "This comment is going to go into the slurm log file, not test.log"
sleep 10
echo Job stopped at `date` >> test.log
```

Here are the resulting message and files after running `sbatch test.sh`

```
> sbatch test.sh
Submitted batch job 2246985
```

```
cat scripts/test.log
Job started at Tue Oct 17 18:36:13 EDT 2023
Hello World!
Job stopped at Tue Oct 17 18:36:23 EDT 2023
cat scripts/slurm-2246985.out
This comment is going to go into the slurm log file, not test.log
```

Slurm options

- Time limit (`-t` or `--time=<time>`) – This must always be specified
  - Formats:
    - `-t 1` for one minute
    - `-t 1:10` for one minute, ten seconds
    - `-t 1:00:00` for one hour
    - `-t 2-1:00:00` for two days, one hour
    - `-t 1-1` for one day, one hour
  - Time limit on jobs on htc is `6-00:00:00`
- Job name (`-J` or `--job-name=<jobname>`)
- CPUs (`-c` or `--cpus-per-task=<ncpus>`)

More info: https://crc.pitt.edu/getting-started/running-jobs-slurm

Command-line arguments in Slurm scripts

Use the `--export` option with `sbatch` to pass command-line arguments:

```
~ > cat script_that_takes_arguments.sh
#!/bin/bash
#SBATCH -t 1:00:00

echo $text | grep -o $keyword


~ > sbatch --export=text='abcde',keyword='de' script_that_takes_arguments.sh
Submitted batch job 2246996
~ > # After done
~ > cat slurm-2246996.out
de
```

Managing Slurm jobs

- Use `squeue -u yourUserName` to check on the status of your jobs

```
> squeue -u jmc108
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           2246988       htc  test.sh   jmc108  R       0:03      1 htc-n52
           2246984       htc     bash   jmc108  R      12:14      1 htc-1024-n1
```

- `ST` is the state of the job (`R` = running, `PD` = pending, `CA` = canceled, `CD` = completed, `TO` = timeout)
- `Time` = how long the job has been running
- Cancel a job with a command like `scancel 123456` (use the ID of the job you want to kill)
- log file
  - every `sbatch` command makes a log file (named like `slurm-123456.out`)
  - the number in the log file matches the slurm job's ID
  - contains stdout and stderr for your script (unless you redirected them)
  - created in the working directory from which you ran the script
  - **LOOK AT THE LOG FILE!!!**

A note about exit status

Recall that when a process ends, it returns an exit status stored in the variable $?

- Exit code 0 means no error
- Any other exit code means error/failure for some reason
- For a **pipeline**, the exit code is the last command's exit code - so errors can hide
- set -e and set -o pipefail refer to nonzero exit codes (for a single command and for a pipeline, respectively)
- Sometimes a script might fail because of a rather "cryptic" triggering of `set -e` or `set -o`, so be careful

Piping in scripts

- Suppose the current step in your pipeline outputs a million lines of text, so you pipe it to a command like `head`, which stops streaming after a few lines
- `head` sends a signal to the last command, telling it to stop streaming the million lines (good!)
- this can cause a non-zero exit status inside the pipeline
- that will trigger `set -o pipefail` and make the script stop running (bad)
- to avoid this, try to re-order your commands
- e.g., instead of `tail -n+2 hugefile.txt | head -n1` do `head -n2 hugefile.txt | tail -n1`