Announcements

Feedback on PAs

- not enough PAs for IE2 → increasing # PAs
- All PA session had its advantages → bring it back
- Small PAs at end of session too → keep it up
- Release solutions for all exercises → you got it

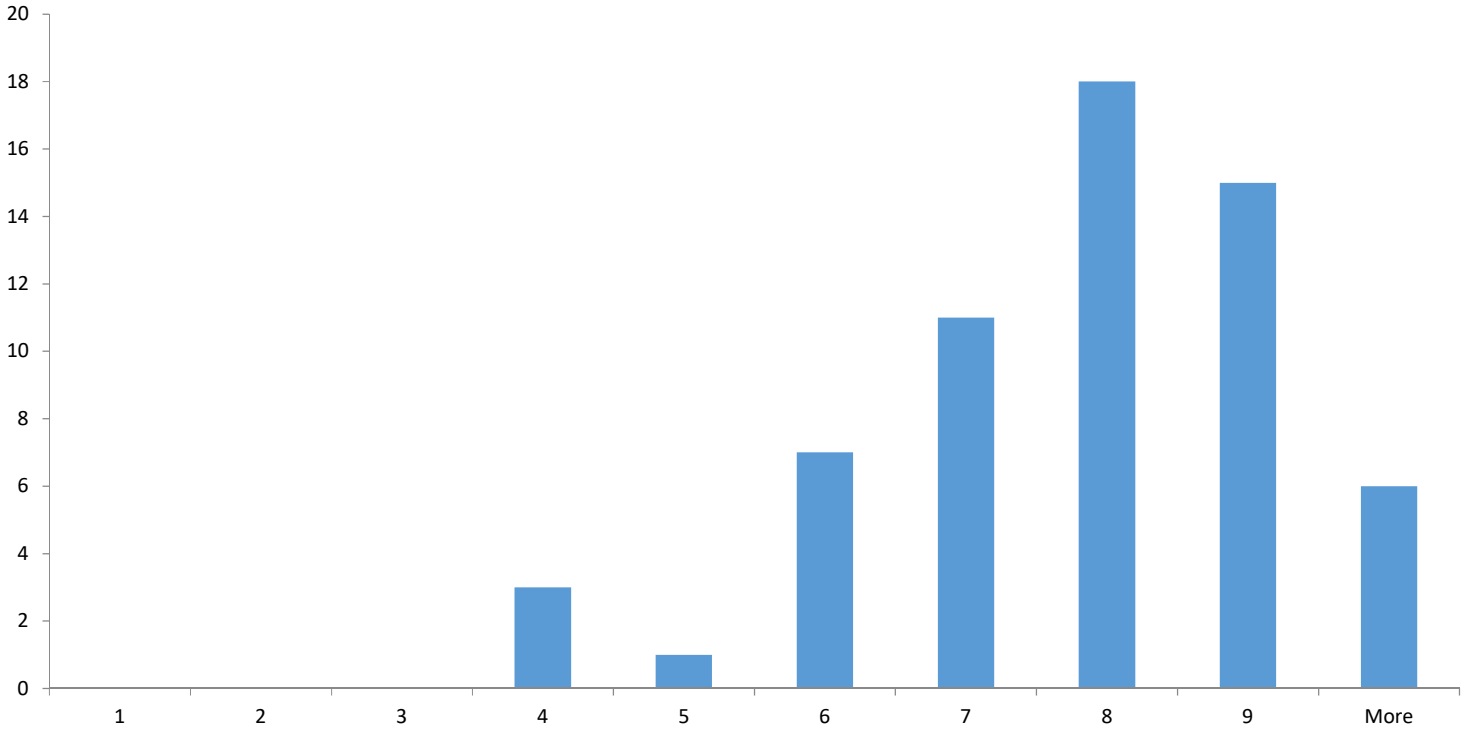
Announcements

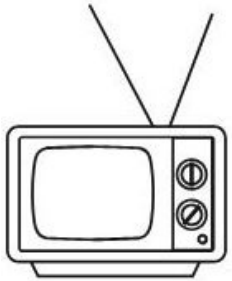
- Tentative grading rubric for the case study
  - Individual per student
  - Please note the technical content requirement

slides			Eloquence			Technical Content		Total
Appropriate Quantity	Grammar, spelling	Visual Quality	Appropriate Pace	Grammar	Clarity	Correctness	Depth	
1	1	1	1	1	1	3	3	12

Announcements

- IE2 grades are in





Previously On... CIS4930

- Redirections Rudiments
  - To / from files
  - Appending / overwriting
- More advanced stuff
  - Merging streams
  - Swapping STDIN & STDOUT
  - Piping
- Unfinished business:
  - PA2b solutions

Topic

Interlude: PA2b – Solutions

These Practice Exercises are meant to help you review for our next IE

Counting bashes

- How do I use piping to count the number of bash interpreters running on my machine?
- Hint: we used the commands `wc` and `grep` in previous slide examples…

```
ps -e | grep bash | wc -l
```

Each step its own log file

#2

- I want to run a multi-steps pipeline of commands but keep the STDOUT at each step in a file out.1, out.2, out.3, out.4 …
- For example, I want to filter out of a dictionary file all words not containing a letter ‘a’, then do the same on the result with words not containing the letter ‘b’, and keep going like this until I have only on STDOUT the words that contain all vowels

```
cat /usr/share/dict/words | grep a | grep e | grep i | grep o
| grep u | grep y
```

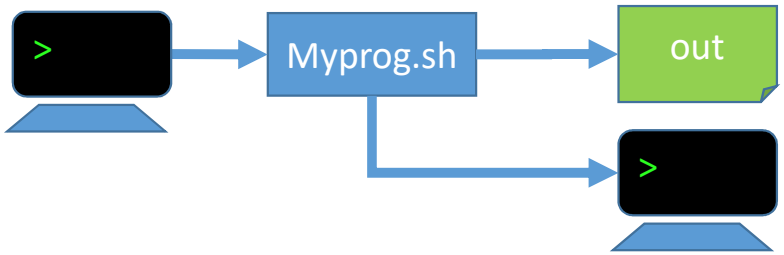
- How do I save each intermediary step’s STDOUT?

```
cat /usr/share/dict/words | grep a | tee out.1 | grep e | tee
out.2 | grep i | tee out.3 | grep o | tee out.4 | grep u |
tee out.5 | grep y
```

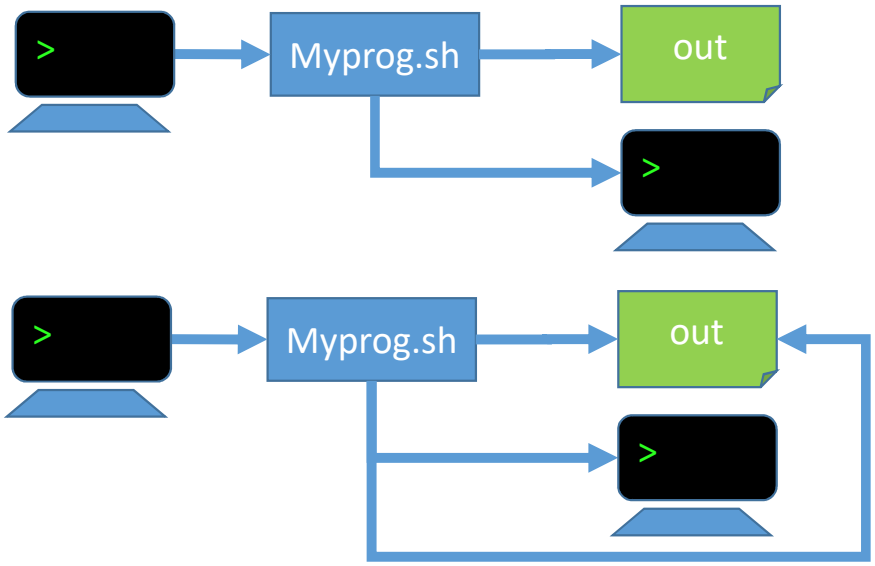
What happens here?


```
./myprog.sh 1>out 3>&1 1>&2 2>/dev/null 1>&3
```

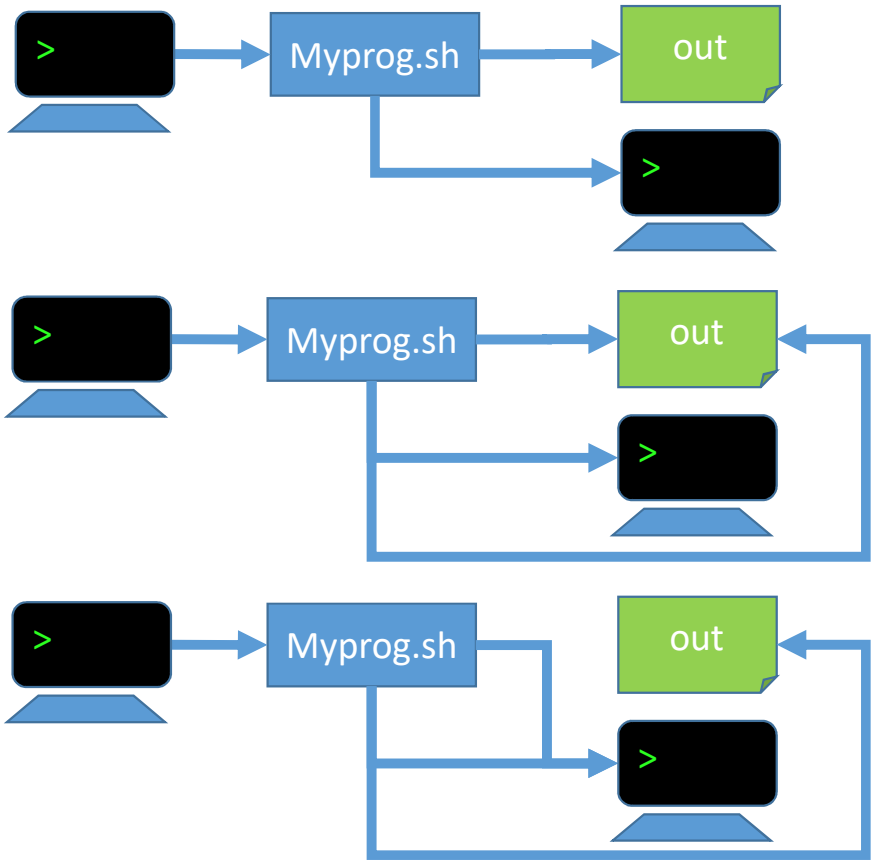
```
./myprog.sh 1>out 3>&1 1>&2 2>/dev/null 1>&3
```



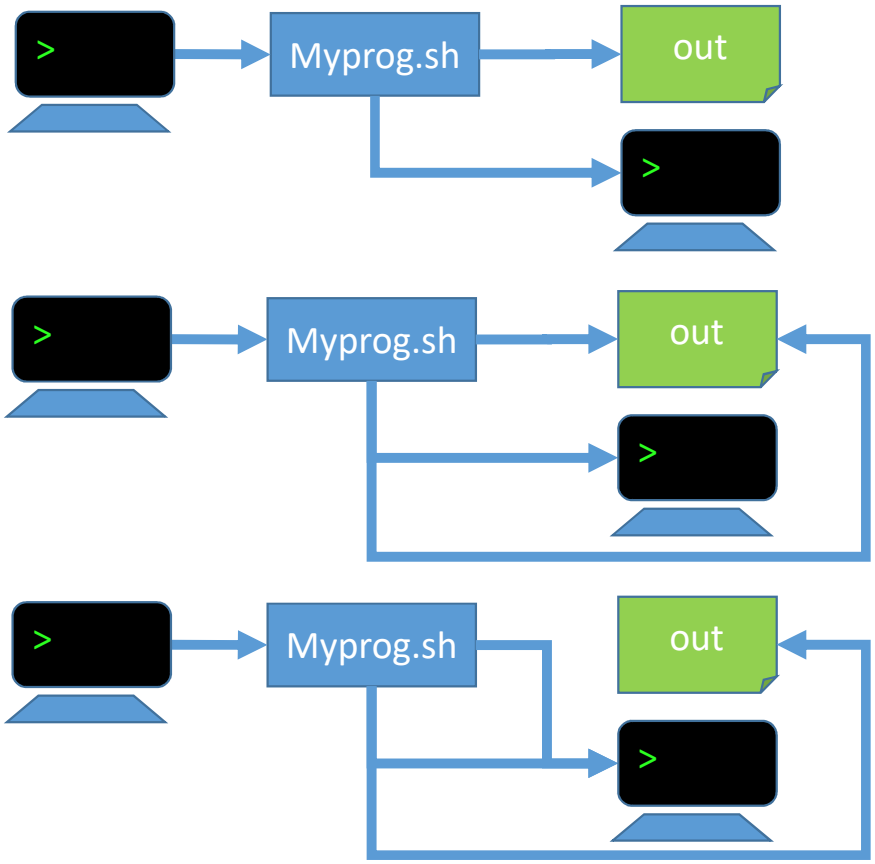
```
./myprog.sh 1>out 3>&1 1>&2 2>/dev/null 1>&3
```

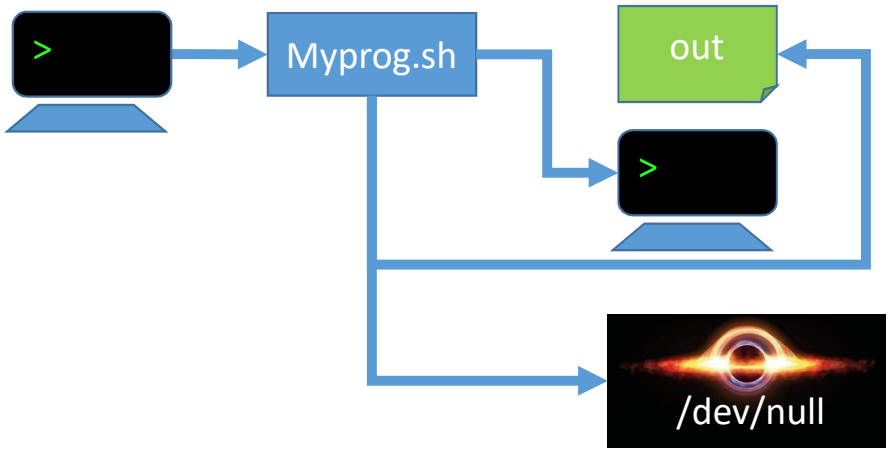


```
./myprog.sh 1>out 3>&1 1>&2 2>/dev/null 1>&3
```

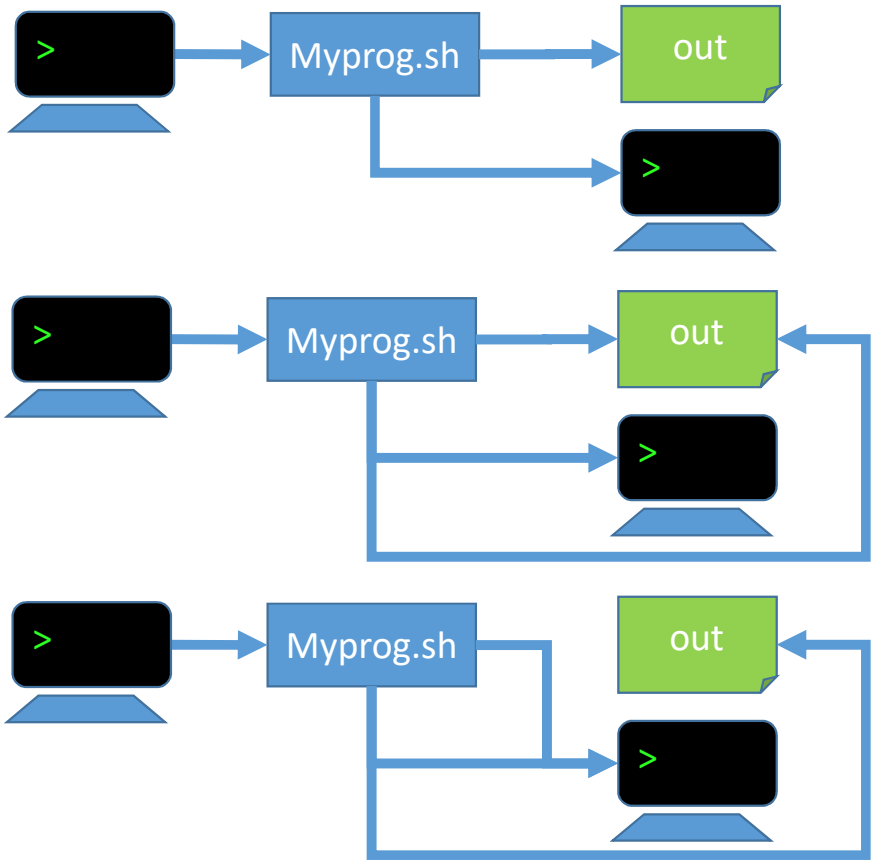


```
./myprog.sh 1>out 3>&1 1>&2 2>/dev/null 1>&3
```





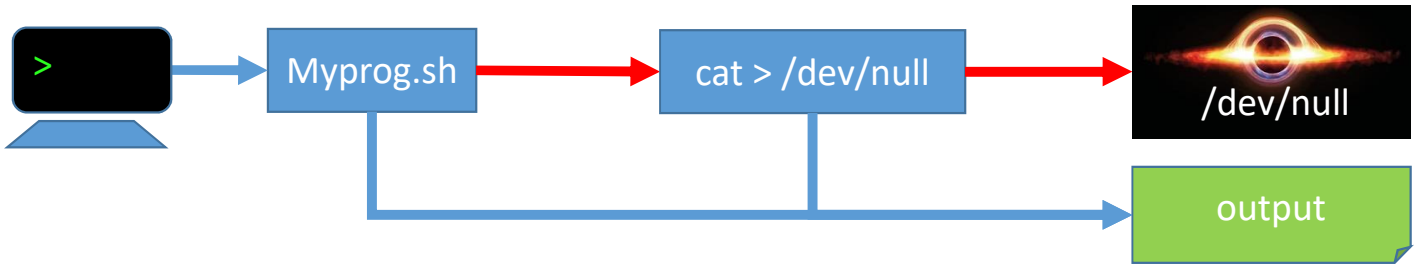
```
./myprog.sh 1>out 3>&1 1>&2 2>/dev/null 1>&3
```



~~Homer's~~ STDERR ~~triple~~ Bypass

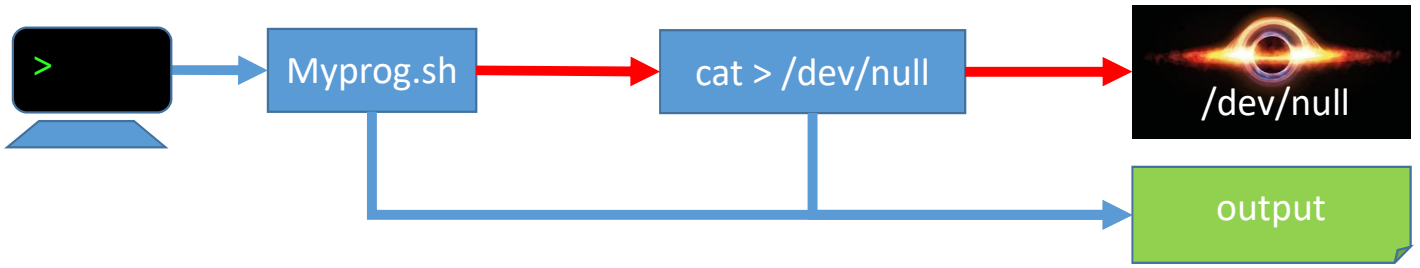
#4

This is what we want:



STDERR Bypass

This is what we want:

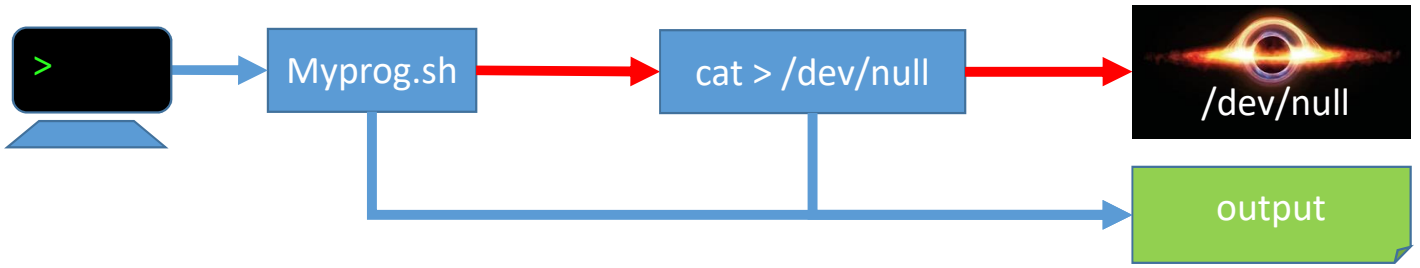



```
$ ./myprog.sh | cat > /dev/null 2> output
this is for STDERR
$ cat output
$
```

Why isn't it working?

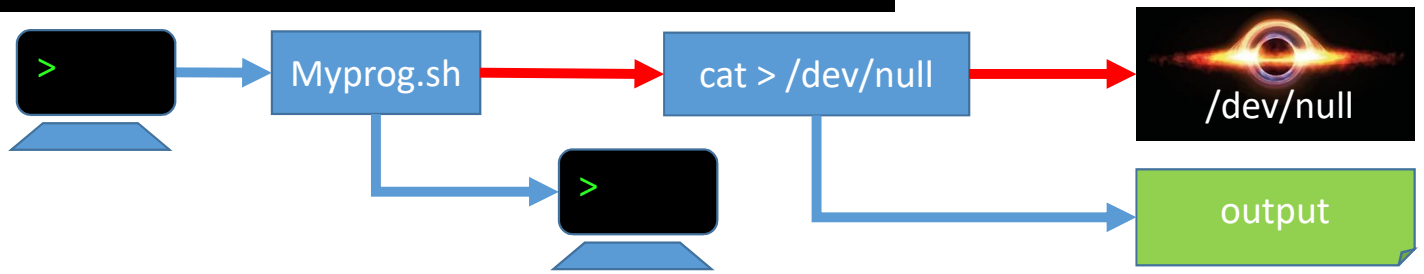
STDERR Bypass

This is what we want:



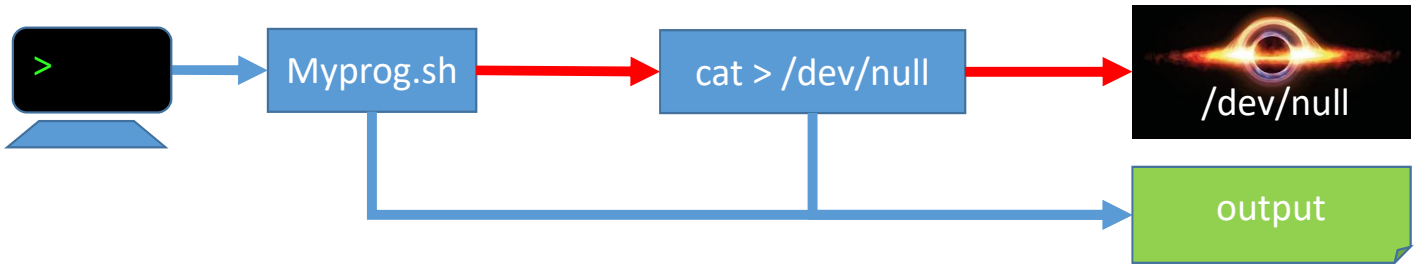

```
$ ./myprog.sh | cat > /dev/null 2> output
this is for STDERR
$ cat output
$
```

Why isn't it working?



STDERR Bypass

This is what we want:

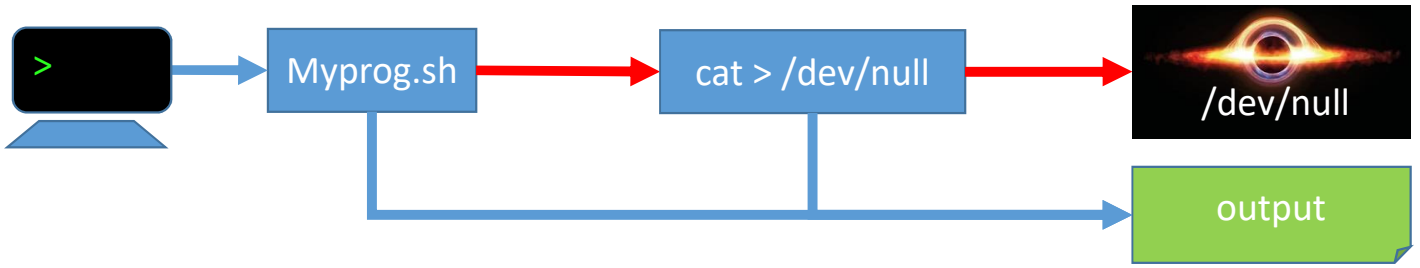



```
$ ./myprog.sh 2> output | cat > /dev/null
$ cat output
this is for STDERR
$
```

Why isn't it working?

STDERR Bypass

This is what we want:

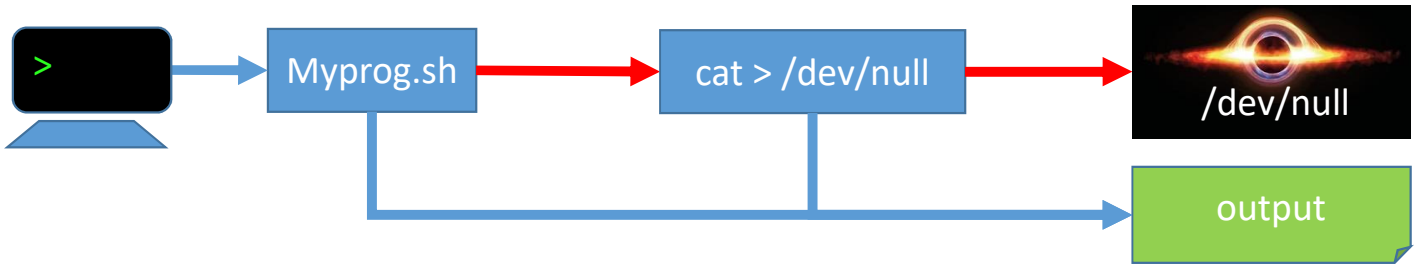



```
$ ./myprog.sh 2> output | cat > /dev/null
$ cat output
this is for STDERR
$
```

Why isn't it working?

STDERR Bypass

This is what we want:

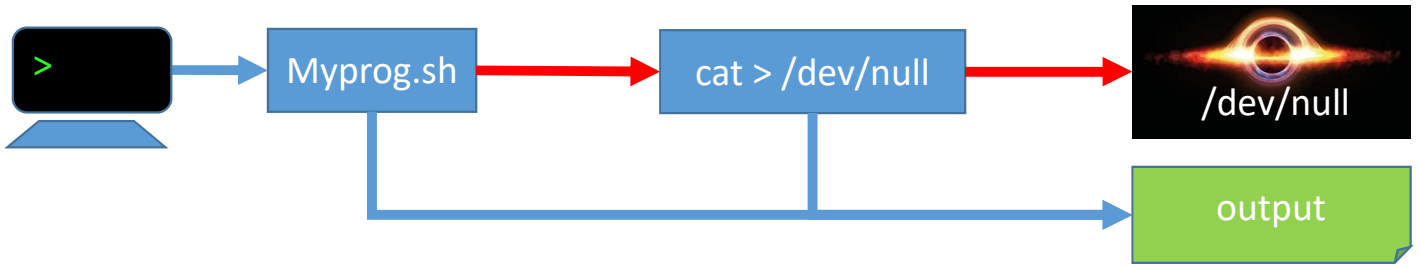



```
$ { ./myprog.sh | cat > /dev/null ; } 2> output
$ cat output
this is for STDERR
$
```

Why isn't it working?

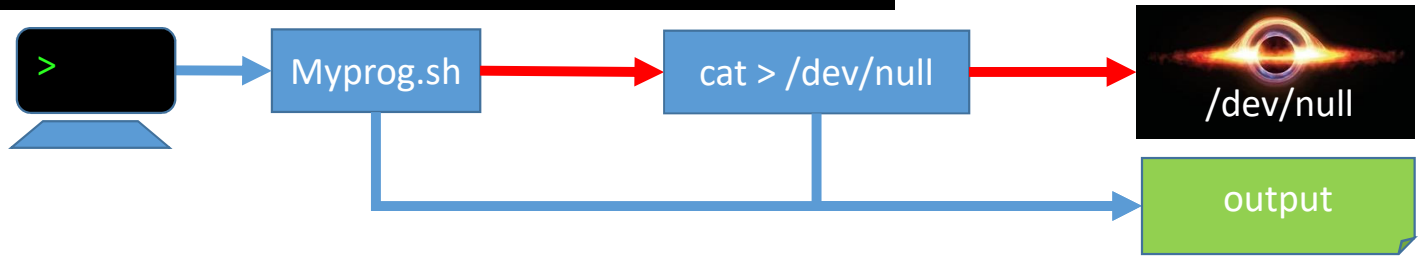
STDERR Bypass

This is what we want:



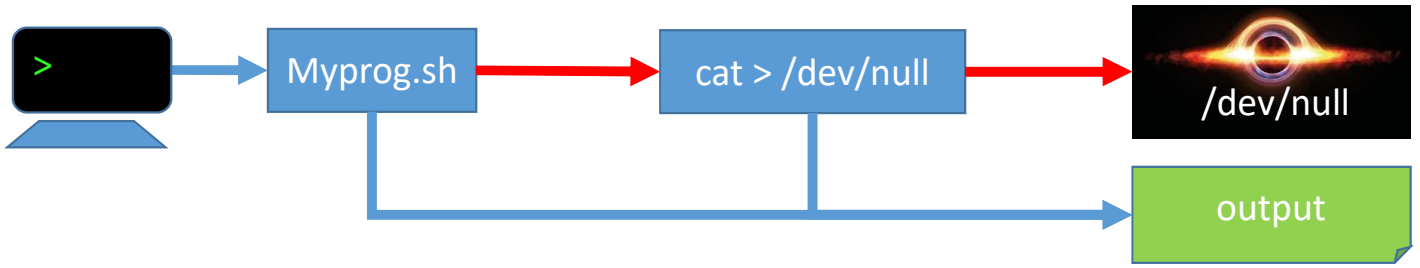

```
$ { ./myprog.sh | cat > /dev/null ; } 2> output
$ cat output
this is for STDERR
$
```

Why is~~n't~~ it working?



STDERR Bypass

This is what we want:

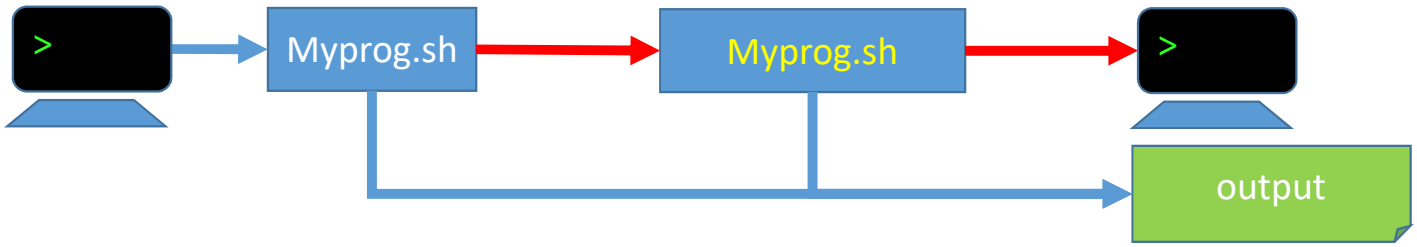



```
$ { ./myprog.sh | cat > /dev/null ; } 2> output
$ cat output
this is for STDERR
$
```

Wait! How are we sure that we get BOTH STDERRs in output?

STDERR Bypass

This is what we want:




```
$ { ./myprog.sh |./myprog.sh ; } 2> output
this is for STDOUT
$ cat output
this is for STDERR
this is for STDERR
```

Topic

M3T2.1
Filters - sort

https://youtu.be/RGEJLK5Cpss

Define what a filter is

- Works line per line on input coming in STDIN
- Apply processing to line
- Output line on STDOUT

Sorting by the entire lines

employees.db →

```
000112222 blue-clearance mid-salary
222334444 red-clearance low-salary
444556666 green-clearance high-salary
666778888 blue-clearance high-salary
888990000 red-clearance mid-salary
```

`sort employees.db`

- Entire line is the key for the sort
- Ascending order

`sort -r employees.db`

- Descending order

Sorting by the a given field

What about sorting by clearance?

```
sort -k 2 employees.db
```

- Field numbering starts at 1
- Default separator == blank space
- This sorts with field #2 until the end of the line

```
sort -k 2,2 employees.db
```

- This sorts only with field #2

How to verify that the outputs are different?

```
sort -k 2    employees.db > out1
sort -k 2,2  employees.db > out2
```

Let's compare

```
diff -s out1 out2
diff -q out1 out2
```

Specifying different field separators

employees2.db →

```
000112222:blue-clearance:mid-salary
222334444:red-clearance:low-salary
444556666:green-clearance:high-salary
666778888:blue-clearance:high-salary
888990000:red-clearance:mid-salary
```

```
sort -k 2,2 employees2.db > out3
```

- `diff -q out2 out3` → they differ
- Sort saw this file as having 1 field per line since there were no spaces

```
sort -k 2,2 -t ':' employees2.db > out4
```

- `diff -q out2 out4` → same!!!!

Topic

M3T2.2
Filters – cut & paste

https://youtu.be/en0RFkr2muc

Basics of cutting

- employees.db & employees2.db from previous slides
- Note that the SSN is always 9 chars long at the beginning of line

```
000112222 blue-clearance mid-salary
222334444 red-clearance low-salary
444556666 green-clearance high-salary
666778888 blue-clearance high-salary
888990000 red-clearance mid-salary
```

```
000112222:blue-clearance:mid-salary
222334444:red-clearance:low-salary
444556666:green-clearance:high-salary
666778888:blue-clearance:high-salary
888990000:red-clearance:mid-salary
```

Basics of cutting (w/ fixed-length fields)

```
cut -c 1 employees.db
```

- Counting starts at 1
- Shows, for each line, only the 1st character for that line

```
cut -c 1,2,3 employees.db
```

- List of characters

```
cut -c 1,2,3,6,7,8,9 employees.db
```

- First 3 chars and last 4

```
cut -c 1-3,6-9 employees.db
```

- Same than above but specifying a range of characters

What if a field does not have a fixed width

Let's extract the clearance field:

```
cut -c 11-24 employees.db
```

- Not exactly what we want ☺
- Fields have varying lengths…

```
cut -d ' ' -f 1-2 employees.db
```

- Numbering of fields starts at one
- We can specify the delimiter with –d
- Ranges are ok 1-3

```
cut -d ' ' -f 1,3 employees.db
```

- Lists are ok 1,2,3

Side note: is the order of columns meaningful?

```
cut -d ' ' -f 1,3 employees.db
```

vs.

```
cut -d ' ' -f 3,1 employees.db
```

- NO, Same output in each case
- Order of selected columns in the file is preserved in the output

```
cut -d ' ' -f 1-2 employees.db
```

vs.

```
cut -d ' ' -f 2-1 employees.db
```

- YES, Error message about invalid decreasing range

What happens if we specify the wrong delimiter

```
cut -d ':' -f 1,3 employees.db
```

- No error messages
- Cut outputs the entire line, unchanged, since it does not find a delimiter
  - Useful for lines that are comments in the file; they are still shown in output
  - Can be useful, can get in your way, it's just the default behavior

```
cut -s -d ':' -f 1,3 employees.db
```

- Stands for suppress
- Ignore lines without the specified delimiters

Now looking into the `paste` filter

Setting up our new files

- `cut -d ' ' -f 1 employees.db > ssn.db`
- `cut -d ' ' -f 2 employees.db > security.db`
- `cut -d ' ' -f 3 employees.db > salary.db`

What does `paste` do?

- `paste` takes 2 files and reconstitutes a multi-column file
- It assumes that the order in each file is "in sync": i.e.,
- Each line in each file has the info for the corresponding employee

```
paste salary.db ssn.db
```

Output delimiter for `paste`

By default, it is a TAB, let's change that:

```
paste -d ':' salary.db ssn.db security.db
```

```
paste -d ' :' salary.db ssn.db security.db
```

- More than 1 delimiter specified (here, a space and a colon)
- Paste uses the 1st one between salary & ssn
- Then uses the 2nd one between ssn & security

We can also do piping with…

cut:

```
cat employees.db | cut -f 1 -d ' '
cat employees.db | cut -f 1,2 -d ' ' | cut -f 2 -d ' '
```

paste:

```
cat ssn.db | paste -d ":" security.db - salary.db
```

- The – stands for STDIN

Interlude: swapping 2 columns w/ cut & paste

Introducing… subshell notation.

```
paste <(cut -d ' ' -f 2 employees.db) <(cut -d ' ' -f 1 employees.db)
```

"A command substitution ($(...)) will be replaced by the output of the command, while a process substitution (<(...)) will be replaced by a filename from which the output of the command may be read. The command, in both instances, will be run in a subshell."

https://unix.stackexchange.com/questions/393349/difference-between-subshells-and-process-substitution

Topic

M3T2.3
Filters - join

https://youtu.be/4Me72elHFs0

Preparing our “DB”

`names.db`

```
Mulder Fox 000112222
Scully Dana 111223333
Anderson Thomas 222334444
Anderson Tiffany 333445555
Cooper Dale 444556666
Cooper Murphy 555667777
Watts Wade 666016666
Cook Samantha 666026666
Flynn Kevin 777889999
McPhearson James 888990000
Snape Severus 999001111
```

`usernames.db`

```
fmuld 000112222
sdana 111223333
neo 222334444
trinity 333445555
dcoop 444556666
murph 555667777
parzival 666016666
art3mis 666026666
clu 777889999
James 888990000
hbp 999001111
```

All files have their lines already sorted by ascending SSN

So that we don’t have to sort everything all the time

Join works only on files that have been sorted by the keys that we want to join with

Basic usage

`join` assumes the 1st field is the key, this is not true for our files

```
join -t ' ' -1 3 -2 2 names.db  usernames.db
```

- -t → specifies the delimiter
- -1 → 1st file key selection
- 3 → key is the 3rd field, assuming delimiter

What does the output look like?

- Key is 1st field in the output
- Combines all fields available in the files in the order the files were specified
- Output is sorted by ascending key as well

Dealing w/ files missing information about some keys

- Removing mcphearsons entry from `names.db`
  - 1 SSN appears in `usernames.db` but not `names.db`

```
join -t ' ' -1 3 -2 2 names.db  usernames.db
```

    - By default, nothing appears about McPhearson
    - By default, we want only the keys that are present in BOTH files and ignore the other ones

- Removing dcoop entry in usernames but keep their entry in names.db
  - Same here now the output skips both mcphearson and dale cooper

→Basic join is the intersection

every key must appear in both files

The -a option

```
join -t ' ' -1 3 -2 2 -a1 names.db  usernames.db
```

- We want all entries in 1st file even if their key is missing from 2nd file
- Cooper dale shows up in output but without a username field since there is none to be found

```
join -t ' ' -1 3 -2 2 -a2 names.db  usernames.db
```

- We want all entries in 2nd file even if their key is missing from 1st file
- Mcphearsons' username shows but not his full name

→ Not just the intersection anymore

→ makes sure all entries from a given file are going to show up in output, even if this means they will have partial information

The -v option (opposite to -a)

Usage: -v1 or -v2 or -v1 -v2

-v1

- Only Entries from 1st file that do not have corresponding entries in 2nd file must be in output
- "Only what is in file 1 and not in file 2"

```
join -t ' ' -1 3 -2 2 -v1 names.db  usernames.db
```

- Only shows line for dale cooper

```
join -t ' ' -1 3 -2 2 -v1 -v2 names.db  usernames.db
```

- All entries that are in one file but not the other (both of the above)

Let's do some piping!

```
join -t ' ' -1 3 -2 2 names.db  usernames.db
     | cut -d ' ' -f 2,3,4
```

- If we want to see only the names and not the key

```
join -t ' ' -1 3 -2 2 names.db  usernames.db
     | cut -d ' ' --complement -f 1
```

- Extract All fields BUT 1

What if we want the username first and names after?

```
join -t ' ' -1 2 -2 3 usernames.db  names.db
     | cut -d ' ' --complement -f 1
```

Topic

Interlude: PA3a

These Practice Exercises are meant to help you review for our next IE.

Timestamps

- We want to write a command line that allows us to append information about the current time/date into a text file.
- We are going to use the `date` tool, and redirections, to append the current date and time into a text file named *`timestamps.log`* which we will assume is in the working directory.
- We will not be assuming that the file exists, but we will make sure that if we use our solution multiple times, then new lines are appended at the end of the file without overwriting previous information.

Improved Timestamps

- Let us get fancy with the format with which the information will be recorded on each line of our text file; Instead of just having the date, we need to have the following information on each line;

  YEAR MONTH DATE DAY HH:MM:SS

- An example of an entry would be;

  2011 February 18 Friday 19:42:00

Hint: *man date* is your friend

The Better Timestamps Bureau

- What about improving our previous command line so that we are able to also append the user's name at the end of the line. The previous entry would now read instead;

  2011 February 18 Friday 19:42:00 smith

Filtering by Users

- Use the previous command line several times so that you get a timestamps.log file with a variety of entries (or use a text editor to make entries up, even easier).

- Feed this log file to 1 or more filters so that you only display the username part of each line.
  - Ensure, with the sort filter, that the list of users is actually sorted
  - Ensure, with the `uniq` filter, that each username is displayed only once
  - Use a filter to count how many users were in the above list

Filtering by Dates

- We want to feed the timestamps.log file through one or more filters so that we end up displaying the list of all months, in alphabetical order, that had entries in the file.
- Each month should be displayed only once.

Joining timestamps w/ usernames

We want to use *join* to combine our *timestamps.log* file

```
2011 February 18 Friday 19:42:00 smith
```

With *usernames.log* that lists usernames, followed by a space, and then the full name.

```
mulder Fox Mulder
scully Dana Scully
murph Murphy Cooper
dale Dale Cooper
```

Start by sorting both files based on the usernames and thus create two new versions named *timestamps-sorted.log* and *usernames-sorted.log*.

- Once you have these, use them in your *join* operation and dump the result in a file named *result-sorted.log*.

- The end result of our combination should show all lines from *timestamps.log* with, at the end, the full name of the user (when available). Please note the following;
  - The order will not be chronologic anymore
  - *join* will put the username first on each line
  - Neither of these is an issue.