

Valgrind, GDB & Penn-Shell M1

A crash course on learning to love seg faults



Table of Contents

1. Valgrind
2. GDB
3. File Descriptors, Redirections & Pipes
4. Penn Shell Milestone 1

Valgrind

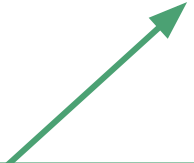
Valgrind

- Debugging tool for detecting memory bugs and leaks
- Basically a program that 'runs' your program, looks for potential memory leaks and reports them
- Why Valgrind?
 - Tells you position of segfaults
 - Usage of non-initialized values
 - Errors in memory usage: double free, wrong memory parameters


Valgrind Usage

```
./valgrind --leak-check=full --show-leak-kinds=all -track-origins=yes [program name]
```

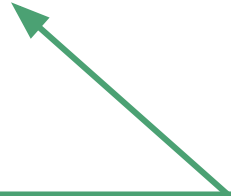
Enable memory leak
detection



Show all “definite, indirect,
possible, reachable” leak
kinds

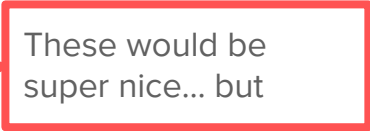


Track origins of
uninitialized values



Example Run

```
$ valgrind ./penn-shredder 2
==151614== Memcheck, a memory error detector
==151614== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==151614== Using Valgrind-3.15.0 and LibVEX; rerun with -for copyright info
==151614== Command: ./penn-shredder 2
==151614==
penn-shredder# sleep
sleep: No such file or directory
==151627==
==151627== HEAP SUMMARY:
==151627== in use at exit: 0 bytes in 0 blocks
==151627== total heap usage: 3 allocs, 3 frees, 1,512 bytes allocated
==151627==
==151627== All heap blocks were freed -- no leaks are possible
==151627==
==151627== For lists of detected and suppressed errors, rerun with: -s
==151627== ERROR SUMMARY: 0 errors from 0 contexts(suppressed: 0 from 0)
penn-shredder# /bin/sleep
/bin/sleep: missing operand
Try '/bin/sleep --help' for more information.
penn-shredder# /bin/sleep 3
Bwahaha ... Tonight, I dine on turtle soup!
penn-shredder# /bin/sleep 1
penn-shredder# cat ^C
penn-shredder# /bin/cat
Bwahaha ... Tonight, I dine on turtle soup!
penn-shredder# ^C
penn-shredder# /bin/cat
^C
penn-shredder#
==151614==
==151614== HEAP SUMMARY:
==151614== in use at exit: 0 bytes in 0 blocks
==151614== total heap usage: 6 allocs, 6 frees, 112 bytes allocated
==151614==
==151614== All heap blocks were freed -- no leaks are possible
==151614==
==151614== For lists of detected and suppressed errors, rerun with: -s
==151614== ERROR SUMMARY: 0 errors from 0 contexts(suppressed: 0 from 0)
```



These would be
super nice... but

Common Errors

- Conditional jump or move depends on uninitialised value(s)
- Invalid read of size 8
- 8 bytes in 1 blocks are definitely lost in loss record 1 of 7
- Process terminating with default action of signal 11 (SIGSEGV)
- ==29== HEAP SUMMARY:
 - ==29== in use at exit: 74,043 bytes in 11 blocks
 - ==29== total heap usage: 13 allocs, 2 frees, 74,115 bytes allocated
 - ==29== LEAK SUMMARY:
 - ==29== definitely lost: 16 bytes in 2 blocks
 - ==29== indirectly lost: 0 bytes in 0 blocks
 - ==29== possibly lost: 0 bytes in 0 blocks
 - ==29== still reachable: 0 bytes in 0 blocks
 - ==29== suppressed: 74,027 bytes in 9 blocks

Example 1. Invalid Write

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
    int i;
    int *a = malloc(sizeof(int) * 10);
    if (!a) return -1; /*malloc failed*/
    for (i = 0; i < 11; i++){
        a[i] = i;
    }
    free(a);
    return 0;
}
```

```
==23779== Invalid write of size 4
==23779==    at 0x400548: main (invalid_write.c:9)
==23779== Address 0x4c30068 is 0 bytes after a block of size 40
alloc'd
==23779==    at 0x4A05E46: malloc (vg_replace_malloc.c:195)
==23779==    by 0x40051C: main (invalid_write.c:6)
```

What's wrong?

- Writing to memory that is not allocated

How to fix?

- Follow line numbers to where it happened
- Check if you allocated space for it
- Really check. Double check
- Usually indexing error

Example 2. Uninitialized Values

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
    int i;
    int a[10];
    for (i = 0; i < 9; i++){
        a[i] = i;

    for (i = 0; i < 10; i++){
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
```

==24599== Conditional jump or move depends on uninitialised value(s)
==24599== at 0x33A8648196: vfprintf (in /lib64/libc-2.13.so)
==24599== by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599== by 0x400567: main (uninitialized.c:11)
==24599== Use of uninitialised value of size 8
==24599== at 0x33A864484B: _itoa_word (in /lib64/libc-2.13.so)
==24599== by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so)
==24599== by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599== by 0x400567: main (uninitialized.c:11)
==24599== Conditional jump or move depends on uninitialised value(s)
==24599== at 0x33A8644855: _itoa_word (in /lib64/libc-2.13.so)
==24599== by 0x33A8646D50: vfprintf (in /lib64/libc-2.13.so)
==24599== by 0x33A864FB59: printf (in /lib64/libc-2.13.so)
==24599== by 0x400567: main (uninitialized.c:11)
==24599==

What's wrong?

- Usage of uninitialized value

How to fix?

- Go through line numbers to find where
- Try to initialize everything, char[], int... give it a default value
- calloc(3) instead of malloc(3) saves you sometimes
 - calloc(3) = malloc + initialize

Example 3. Memory Leaks

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
    int i;
    int *a;

    for (i=0; i < 10; i++){
        a = malloc(sizeof(int) * 100);
    }
    free(a);
    return 0;
}
```

```
==24810== HEAP SUMMARY:
==24810==    in use at exit: 3,600 bytes in 9 blocks
==24810== total heap usage: 10 allocs, 1 frees, 4,000 bytes allocated
==24810==
==24810== 3,600 bytes in 9 blocks are definitely lost in loss record 1 of 1
==24810==    at 0x4A05E46: malloc (vg_replace_malloc.c:195)
==24810==    by 0x400525: main (mem_leak.c:9)
==24810==
==24810== LEAK SUMMARY:
==24810==    definitely lost: 3,600 bytes in 9 blocks
==24810==    indirectly lost: 0 bytes in 0 blocks
==24810==    possibly lost: 0 bytes in 0 blocks
==24810==    still reachable: 0 bytes in 0 blocks
==24810==    suppressed: 0 bytes in 0 blocks
```

What's wrong?

- Memory is leaked: we did not free enough

How to fix?

- Line numbers
- 1 alloc = 1 free rule
- When we exit(2) because of error, we want to free(3) everything too
- Custom exit function could be helpful

GDB

What is GDB?

- A scary program
- GNU Project Debugger
 - GNU = GNU's not Unix! (recursive acronym)
- You can see what's going on "inside" a program as it executes
- Supports Assembly, C, C++, D, Fortran, Go, Rust, etc.

Using GDB

- `-g` flag in Makefile for compiling

Type in your shell:

- a) `gdb penn-shredder`
- b) `gdb --args penn-shredder 3`
- c) `gdb`
 - `(gdb) file penn-shredder`
 - `(gdb) set args 3`
 - `(gdb) run < in.txt`
 - `(gdb) help [command]`

Walking through code

Command	Shortcut	Description
start		<ul style="list-style-type: none">● Start from beginning and stop there
★ run	r	<ul style="list-style-type: none">● Start and run program from beginning
★ continue	c	<ul style="list-style-type: none">● Run until program exits*
★ step	s	<ul style="list-style-type: none">● Run until next line*<ul style="list-style-type: none">○ Steps <i>into</i> a function
★ next	n	<ul style="list-style-type: none">● Run until next line <i>in the current function</i> is reached / returns*<ul style="list-style-type: none">○ Steps <i>over</i> a function
finish	fin	<ul style="list-style-type: none">● Run until the current function finishes*

*or until next breakpoint


★ = command I use often


Walking through code: example

```
int main(int argc, char* argv[]) {  
    int n = 3;  
    int fib = fibonacci(n);  
    fprintf(stderr, "Fibonacci: %d\n", fib);  
  
    return 0;  
}
```

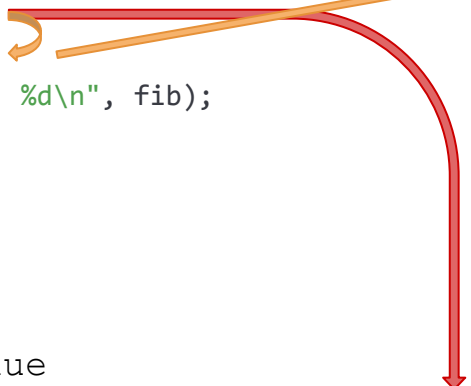
```
int fibonacci(int n) {  
    int t1 = 0;  
    int t2 = 1;  
    int next = 0;  
  
    if (n == 1) {  
        return t1;  
    } else if (n == 2) {  
        return t2;  
    }  
  
    for (int i = 3; i <= n; i++) {  
        next = t1 + t2;  
        t1 = t2;  
        t2 = next;  
    }  
  
    return next;  
}
```

 = continue

 = step

 = next

breakpoint 



Where am I in the code?

Command	Description
<code>layout src</code>	<ul style="list-style-type: none">• Changes the window layout to show source code
<code>refresh / ref</code>	<ul style="list-style-type: none">• Refreshes the window in case it looks weird
<code>Ctrl-x + a</code>	<ul style="list-style-type: none">• Close this window layout view
<code>list / l</code>	<ul style="list-style-type: none">• Show some lines of source code before/around current
★ <code>backtrace / bt / where</code>	<ul style="list-style-type: none">• Displays the call stack
★ <code>frame [number]</code>	<ul style="list-style-type: none">• Selects and inspects a specific stack frame

Breakpoints (b/break)

Command	Description
★ b [filename:]function b [filename:]linenum	<ul style="list-style-type: none">● Sets a breakpoint at the beginning of a function or at a specific line number
★ info breakpoints info b	<ul style="list-style-type: none">● Lists all breakpoints w/ status and conditions
disable [bnum] enable [bnum]	<ul style="list-style-type: none">● Disable or enable a specific breakpoint
★ delete [bnum] d [bnum]	<ul style="list-style-type: none">● Deletes a specific breakpoint● Deletes all if breakpoint num isn't specified
clear [filename:]function clear [filename:]linenum	<ul style="list-style-type: none">● Removes breakpoints in a specific function or at a specific line number

Printing things (p/print)

Command	Description
★ p var	<ul style="list-style-type: none">Prints the value of a variable
p/x var	<ul style="list-style-type: none">Prints the value, in hex (might be useful in PennOS)
p var.field	<ul style="list-style-type: none">Prints a field of a struct
p var->field p (*var).field	<ul style="list-style-type: none">Prints a field of a struct pointer
★ p head.next->next->data	<ul style="list-style-type: none">Example of printing data in a linked list
p *arr[@len]	<ul style="list-style-type: none">Prints the elements of an array, up to the specified length
p var = value	<ul style="list-style-type: none">Sets a different value to a variable

Inspection

Command	Description
★ info args	<ul style="list-style-type: none">• Displays args of the current function
info locals	<ul style="list-style-type: none">• Displays local variables in the current function
info variables [regex]	<ul style="list-style-type: none">• Lists all global and static variables + their data types• Can filter using regex
info functions [regex]	<ul style="list-style-type: none">• Displays all functions in the program• Can filter using regex
ptype [expression]	<ul style="list-style-type: none">• Shows the data type of the given expression• Can display the definition of a type (useful for structs)
★ watch [expression]	<ul style="list-style-type: none">• Stops program whenever value of expression changes• Ex: watch foobar if foobar > 3

Demo: Stack

stack.c

```
typedef struct Node {
    int data;
    struct Node *below;
} Node;

void push(Node *top, int data) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    *newNode = (Node){data, top};
    top = newNode;
}

Node *getMaxNode(Node *top) {
    if (top && top->below && top->data < top->below->data) {
        return getMaxNode(top->below);
    }
    return top;
}
```

```
int main() {
    Node *top = NULL;
    push(top, 1);
    push(top, 3);
    push(top, 2);

    // Results
    Node *maxNode = getMaxNode(top);
    printf("Max: %d\n", maxNode->data);
    return 0;
}
```

Penn Shell-specific debugging

Command	Description
★ <code>signal [signal]</code>	<ul style="list-style-type: none">• Sends a signal (e.g. SIGINT)• Useful to test Ctrl + C, Ctrl + Z, etc
<code>shell [cmd]</code>	<ul style="list-style-type: none">• Executes a command as if you were in bash
<code>shell ps j</code>	<ul style="list-style-type: none">• Lists process(es) info w/ job format output
<code>shell kill -9 <pid></code>	<ul style="list-style-type: none">• Sends a SIGKILL to a specific process (non-ignorable)
<code>kill</code>	<ul style="list-style-type: none">• Kill the program being debugged
<code>set follow-fork-mode [parent child]</code>	<ul style="list-style-type: none">• After a fork, follow the child or parent process• (parent by default)
<code>shell ls -l /proc/<pid>/fd</code>	<ul style="list-style-type: none">• List a process' open fd's

Very fun gdb things

- You can recompile within gdb
 - Just type ``make`` into the gdb shell!
- `gdb -tui`: start gdb with a textual interface (same as running ``layout src``)

More commands

- `set logging on`: log debugging session to show (flex?) to others
- `set pretty array on`: pretty array printing
- `thread apply all bt`: see frames of all threads

And a lot more [here](#) + real life GDB debugging example/walkthrough [here](#)

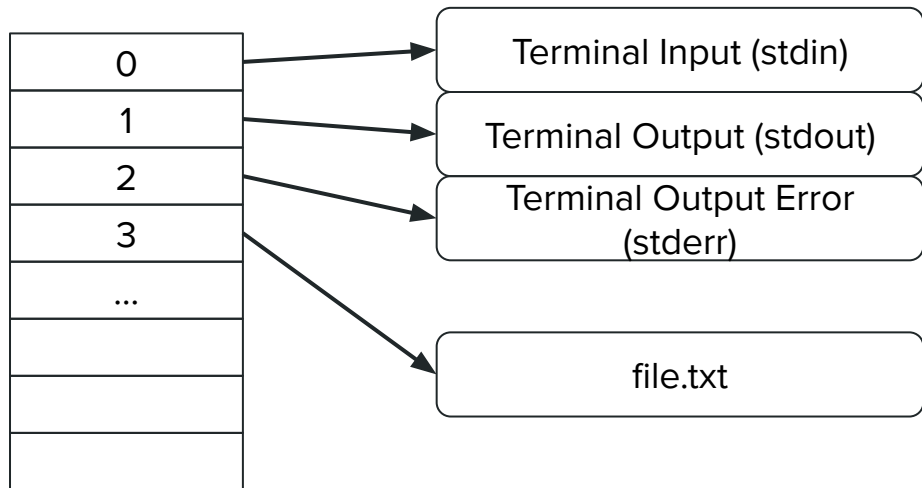
Other Cool GDB Things

Command	Description
<code>disassemble / disas</code>	<ul style="list-style-type: none">● View assembly instructions of the current function
<code>b [filename:]linenum condition</code>	<ul style="list-style-type: none">● Make a breakpoint with an associated condition● e.g. <code>b 73 i > 4 && i % 2 == 0</code>
<code>call</code>	<ul style="list-style-type: none">● Calls a function immediately● Helpful for on-the-fly behavior probing
<code>until, advance, jump, etc.</code>	<ul style="list-style-type: none">● Even more ways to step through your code
<code>python</code>	<ul style="list-style-type: none">● Yes you can do python scripting in GDB
<code>quit / q / Ctrl-D</code>	<ul style="list-style-type: none">● Fixes your bugs instantly● Can touch grass

File Descriptors, Redirections & Pipes

File Descriptor

- Unique id that refers to a file
- Type int
- `read(2)` and `write(2)`
- `open(2)` and `close(2)`
 - Open with unique permissions
 - Read only, write only, read&write, etc
- **Each process has unique file descriptor table**
- 0, 1, 2 reserved for `stdin`, `stdout`, `stderr`



Quick Example

- `read(STDIN_FILENO, buf, 30);`
 - Reads from terminal input and stores to buffer
- `write(STDERR_FILENO, "error message\n", 15);`
 - Write to terminal output error
- `write(STDIN_FILENO, "trying to write\n", 17);`
 - Error. STDIN is "read only"
- `open("file.txt", O_WRONLY | O_CREAT, 0644);`
 - Open file called "file.txt" with "write only" permissions, create if doesn't exist. Also, give 0644 permissions
 - 0644 is an octal integer, each digit after 0 represents **user, group, and other** permissions
 - 6 (read & write) user permission, 4 (read) group permission, 4 (read) other permission

Redirections

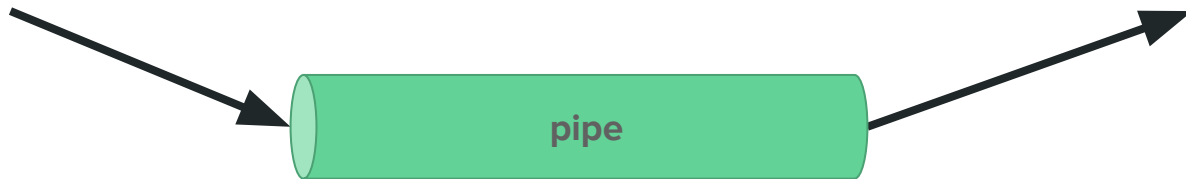
- Redirect a file descriptor to point to some other file!
- `dup2(int oldfd, int newfd)`
 - Whatever file that was pointed to by **oldfd** is now pointed to file pointed to by **newfd**
- `dup2(newfd, STDIN_FILENO)`
 - Redirect **STDIN** to **newfd**. What does this mean?
 - Anything that was supposed to go to **stdin**, which was terminal input, will now go to **newfd**
- `dup2(newfd, STDOUT_FILENO)`
 - Redirect **STDOUT** to **newfd**. What does this mean?
 - Anything that was supposed to be outputted to **stdout**, will now be outputted to **newfd**

Pipes

- FIFO data structure with a read end and write end
 - Picture a pipe with water flowing into (write end) and out of (read end)
- `pipe(2)` system call. `pipe(int pipefd[2])`
 - Creates the pipe data structure pointed to by `pipefd`
 - `pipefd[0]` = read-end, `pipefd[1]` = write-end

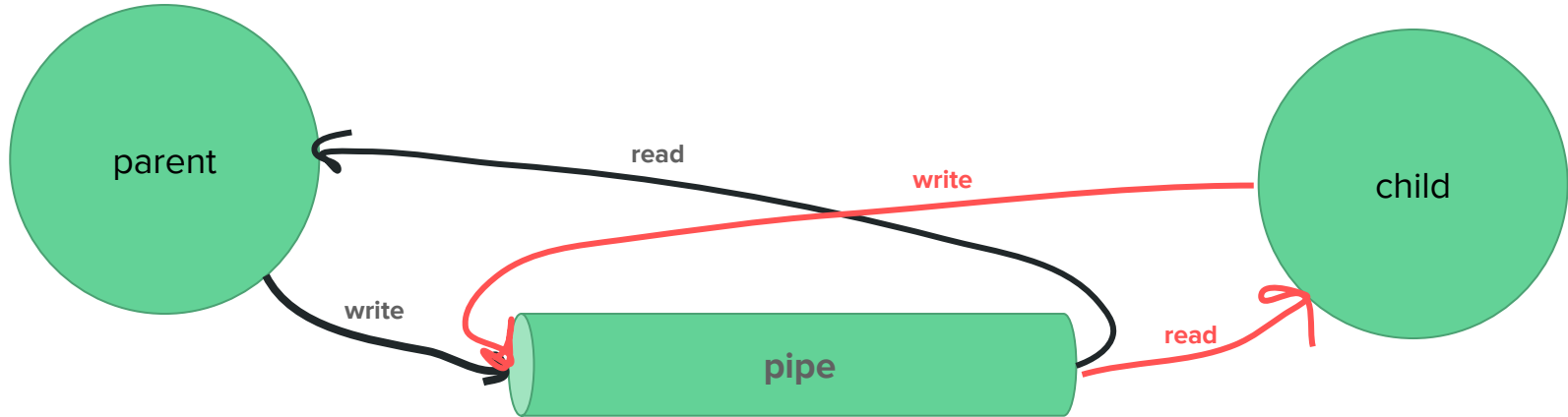
```
write(pipefd[1], "stuff", 6);
```

```
read(pipefd[0], buf, 6);
```



Pipes and processes

- File Descriptor table is “shared” among processes
 - → pipes are shared!!!!
- Child processes has its own copy of each pipe end



Some tips

DRAW! It is easier to visualize what points where.

READ! The system calls related to file descriptors. `open(2)`, `close(2)`, `dup2(2)`, `pipe(2)`

READ CAREFULLY! The man pages for above. Really know what's going on.

E.g. What happens when we `fork(2)` after `pipe(2)`?

What happens if we `close(2)` in a child process?

Penn-Shell M1

Quick Recap of Shredder

```
while(true) {
    write(prompt);
    char cmd_string[];
    int bytes = read(cmd_string);
    struct parsed_command cmd;
    parse_command(cmd_string, cmd);
    pid_t pid = fork();
    if (child){
        exec(cmd);
    }
    else {
        wait()
    }
}
```

- Simple write(), read(), fork(), exec() loop
- Signal handling
 - SIGINT, SIGALRM

Things to add for Shell M1

- Redirections

- `cat < file`
- `ls -l > file`

- Pipes

- `history -1000 | grep ssh`
- `cat < file | wc | cat >> file2`

- You want to fork a child for each command in the pipeline!

```
while(true) {
    write(prompt);
    read(cmd_string);
    parse_command(cmd_string, cmd);
    create appropriate # of pipes
    pipe(pipes)
    for (i = [0, num_command]) {
        pid_t pid = fork();
        if (child){
            redirect if needed
            connect pipes
            exec(cmd);
        }
    }
    waitpid(for all children);
}
```

Things to add for Shell M1

- Redirections

- `cat < file`
- `ls -l > file`

- Pipes

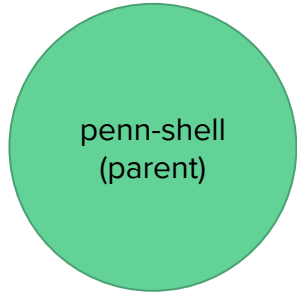
- `history -1000 | grep ssh`
- `cat < file | wc | cat >> file2`

- You want to fork a child for each command in the pipeline!

```
while(true) {
    write(prompt);
    read(cmd_string);
    parse_command(cmd_string, cmd);
    create appropriate # of pipes
    pipe(pipes)
    for (i = [0, num_command]) {
        pid_t pid = fork();
        if (child){
            redirect if needed
            connect pipes
            exec(cmd);
        }
    }
    waitpid(for all children);
}
```

Example Run

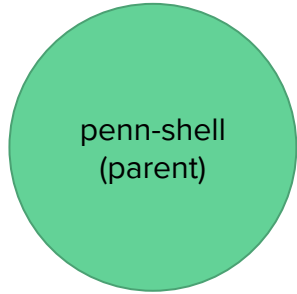
```
cat < file.txt | grep comrade | wc -l -c > out.txt
```



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]

Example Run

```
cat < file.txt | grep comrade | wc -l -c > out.txt
```

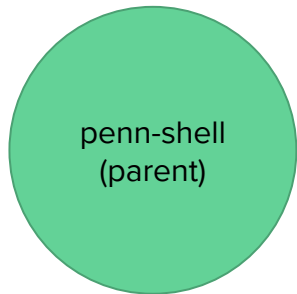


0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



Example Run

```
cat < file.txt | grep comrade | wc -l -c > out.txt
```



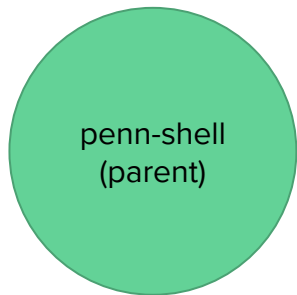
0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]

Example Run

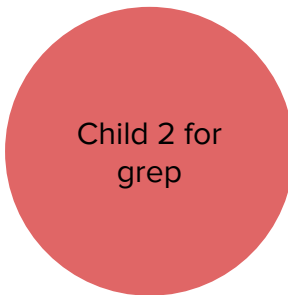
```
cat < file.txt | grep comrade | wc -l -c > out.txt
```



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



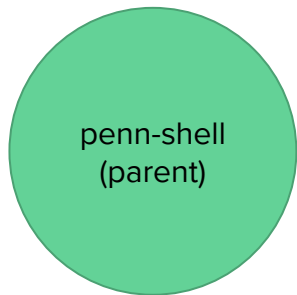
0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]

Example Run

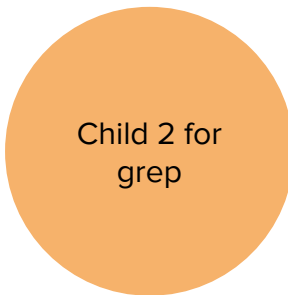
```
cat < file.txt | grep comrade | wc -l -c > out.txt
```



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]

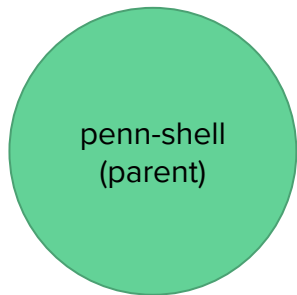


0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]

“Redirect if needed!”

Example Run

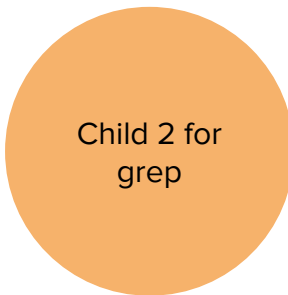
```
cat < file.txt | grep comrade | wc -l -c > out.txt
```



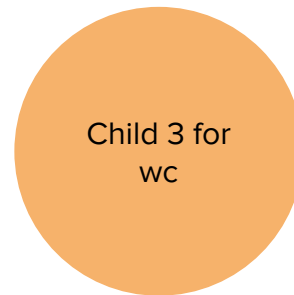
0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]

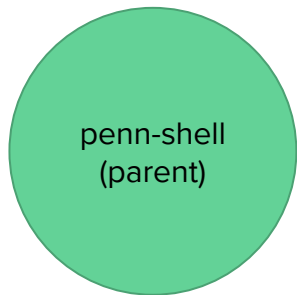


0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]

“Redirect if needed!”

Example Run

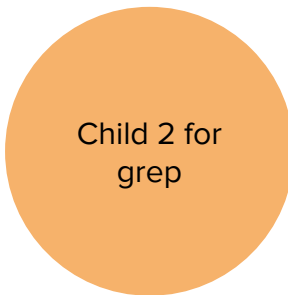
```
cat < file.txt | grep comrade | wc -l -c > out.txt
```



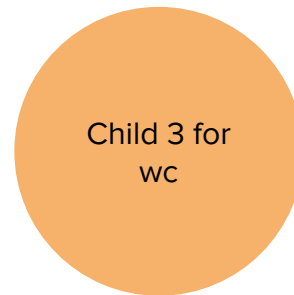
0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



0: file.txt
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]



0: stdin
1: stdout
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]

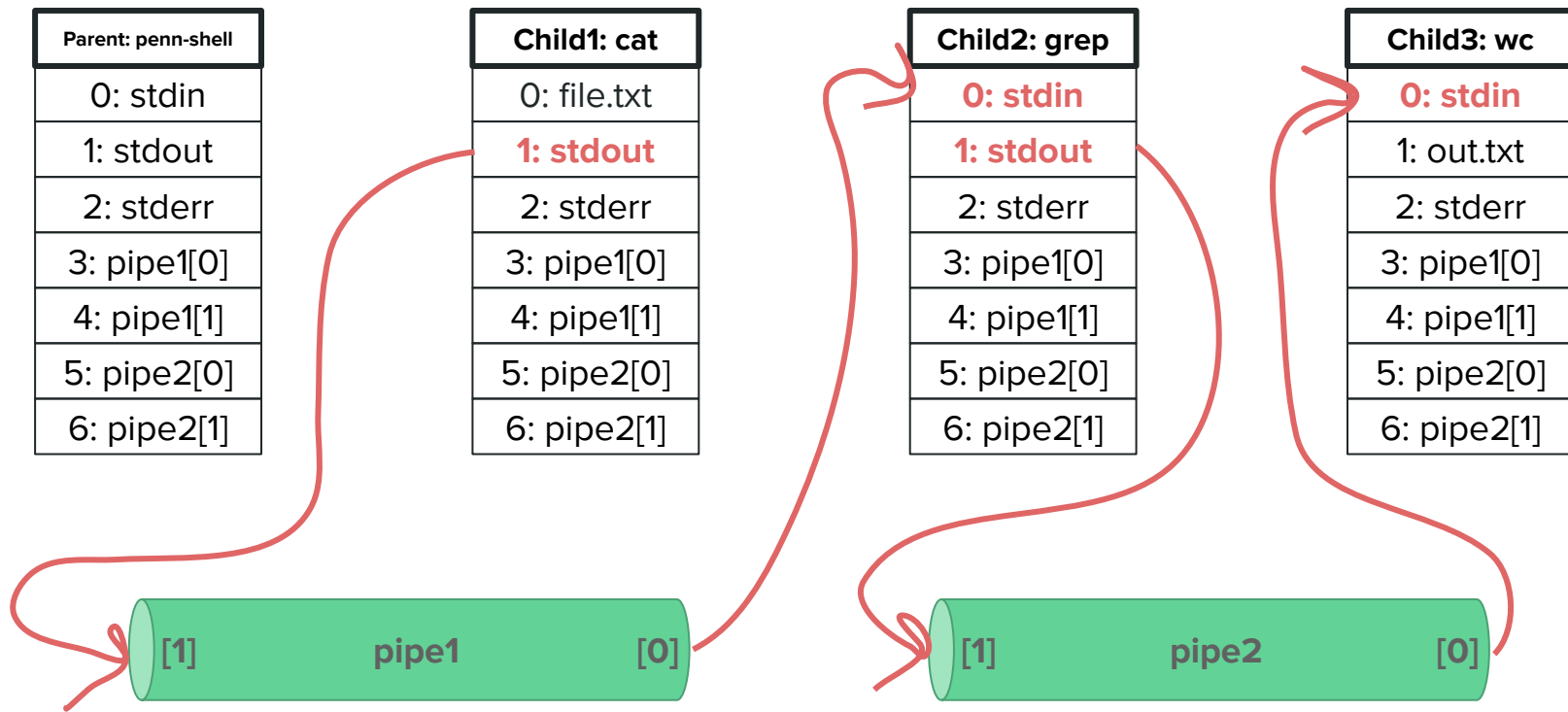


0: stdin
1: out.txt
2: stderr
3: pipe1[0]
4: pipe1[1]
5: pipe2[0]
6: pipe2[1]

“Connect children with pipes!”

Example Run

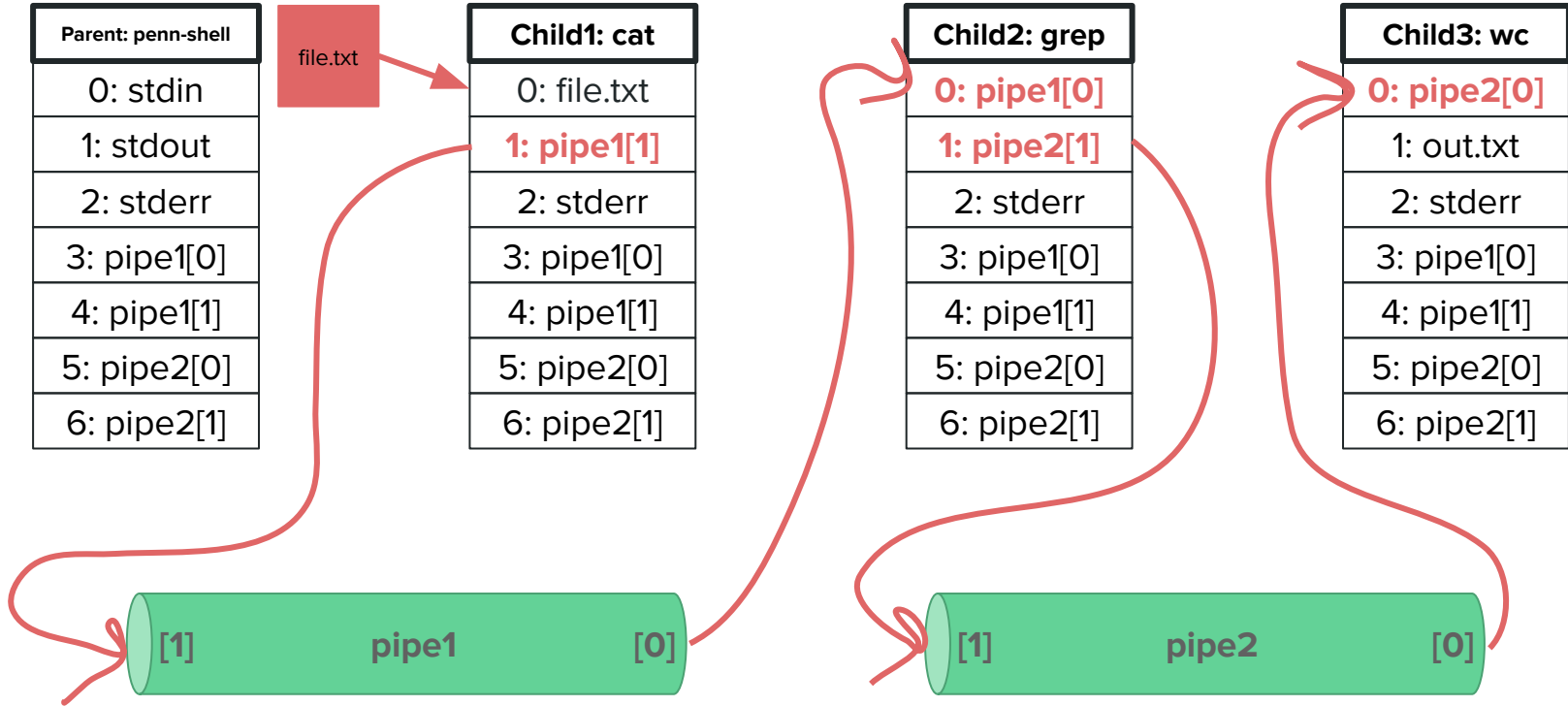
```
cat < file.txt | grep comrade | wc -l -c > out.txt
```



Example Run

Content from “file.txt” will travel through pipes from Child 1 to 3
Each child will execute according to data they read from pipe/file

```
cat < file.txt | grep comrade | wc -l -c > out.txt
```



Example Run

```
root# cat < file.txt | grep comrade | wc -l -c > out.txt  
root#
```

Read from file.txt, output to pipe 1

Read from pipe 1, get lines with the word “comrade”, output to pipe 2

Read from pipe 2, get line count and character count of those lines, output to out.txt

file.txt

Comrade, in the spirit of camaraderie and unity, I extend my heartfelt greetings to you, my esteemed **comrade**.

Together, as **comrades**, we navigate the complex tapestry of life, facing challenges with unwavering solidarity.

Comrade, let us forge ahead, shoulder to shoulder, in pursuit of a brighter tomorrow.

As **comrades**, our shared ideals bind us in a bond that transcends the ordinary, creating a harmonious symphony of collective aspirations.

Comrade, may our journey be marked by mutual support and the enduring strength that comes from the fellowship of like-minded souls.

out.txt

```
3 358
```

Wrap up

- Get into groups for penn-shell!
- Draw Pipe/Redirection diagrams
- Shredder Peer Review due end of the week!
- Milestone due Wednesday Feb 14
- OH for remaining time
- Any questions?