Scribe Notes for CS61: 11/14/13
By David Becerra and Arvind Narayanan

Pipelines, Forks, and Shell

Anecdote on Pipelines:

Anecdote 1: In 1964, Bell Labs manager Doug McIlroy sent a memo stating that programs should be coupled together like gardenhoses. Not only was this memo foundational to the creation of unix, but it was the idea behind pipelines.

Anecdote 2: In one of the *A Programming Pearls* books by John Bentley, there was an article by by Don Knuth. In it Don stated: The problem with programming: programming needs to be like writing documents. Don challenged Jon to come up with a problem that he would implement in his literate programming style

This was the problem:

Given a text file and an integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency – (similar to Heavy Hitters)

Don wrote a BIG PROGRAM (pages of code).
Implementation Ideas:
Could use a hash table for every word, keep links from hashtable to array of words, sort array in terms of frequency, output

Don invented a hash-trie in his implementation. Code was sent to be reviewed by Doug McIlroy, who came up with the idea of plumbing programs together (as we saw earlier).

Doug wrote a Unix shell script of six lines that did the same thing as Knuth's program did in thousands of lines.

(1) tr –cs A-Za-z '
' | (takes everything that's not a letter and turns it into a newline, ignoring punctuation)
(2) tr A-Z a-z | (transliterate uppercase to lower case)
(3) sort | (sorts the input file, 'industrial strength': knows how to handle files that are too big for memory) ("hi hello hi hi bye" => "bye\nhello\nhi\nhi\nhi")
(4) uniq –c | (takes consecutive lines of the same string, outputs only one copy of each line, preceded by the count) ("bye\nhello\nhi\nhi\nhi" => "1 bye\n1 hello\n3 hi")
(5) sort –rn | (sorts in reverse, numerically)
(6) head –n <NUMLINES> (keeps the top NUMLINES number of lines)

So the moral is pipelines are really awesome and powerful! "This is why pipelines".

myecho.c and runmyecho.c

myecho.c: prints myecho's process id (pid) and prints all its arguments starting with arg. 0 which is always the name of the program (i.e. "./myecho").

```
#include "helpers.h"

int main(int argc, char* argv[]) {
        fprintf(stderr, "Myecho running in pid %d\n", getpid());
        for (int i = 0; i != argc; ++i)
        fprintf(stderr, "Arg %d: \"%s\"\n", i, argv[i]);
}
```

---

runmyecho.c: runs myecho

```
#include "helpers.h"

int main(void) {
        char* args[] = {
        "./myecho", // argv[0] is the string used to execute the program
        "Hello!",
        "Myecho should print these",
        "arguments.",
        NULL
        };
        fprintf(stderr, "About to exec myecho from pid %d\n", getpid());

        int r = execv("./myecho", args);

        fprintf(stderr, "Finished execing myecho from pid %d; status %d\n",
        getpid(), r);
}
```

How does runmyecho call myecho? Essentially, a process is forked off, contents of that memory space are replaced by a brand new process image that 's taken by executing the new program; every new program is created as a fork of an existing program. This is all done with the exec system call.

execv - replaces current program with the program in first argument (which should point to a a file name).

execvp - runs a program that it searches for on disk; again, entire memory space of the process is replaced

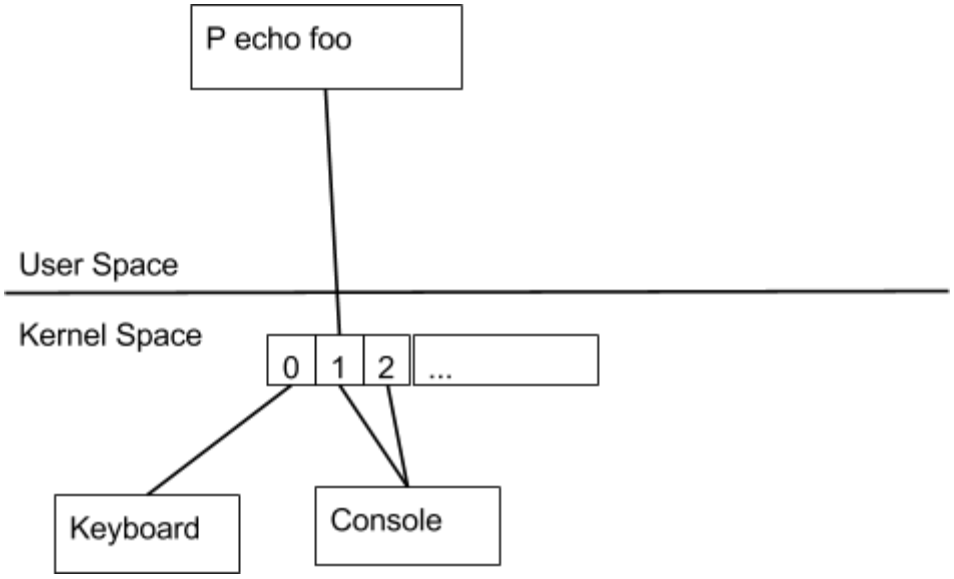
Alarm bells? Where are the arguments?
Printing out the argument's value and its pointer (before and after the exec) in myecho and runmyecho shows that the operating system is copying the arguments from one place in memory to another (from the globals read-only data portion of process memory space) to the stack (above the initial stack pointer, so unused). In addition, the arguments are laid out right next to one another in memory. Basically, the OS is smashing together arguments and putting them on the stack.
What isn't erased?
File descriptors. Memory and environment are replaced but the file descriptors are copied, not erased. For example, myecho inherits stdout from runmyecho. Memory (except for the arguments) are replaced. We cannot guarantee the memory addresses of the args will be conserved or not used, so OS must collect and consolidate all the arguments together.

Argv[0] – how the shell found the program; when runmyecho calls myecho, completely erases itself, replaced by the full copy of myecho
Memory is replaced, file descriptors are inherited (not replaced)

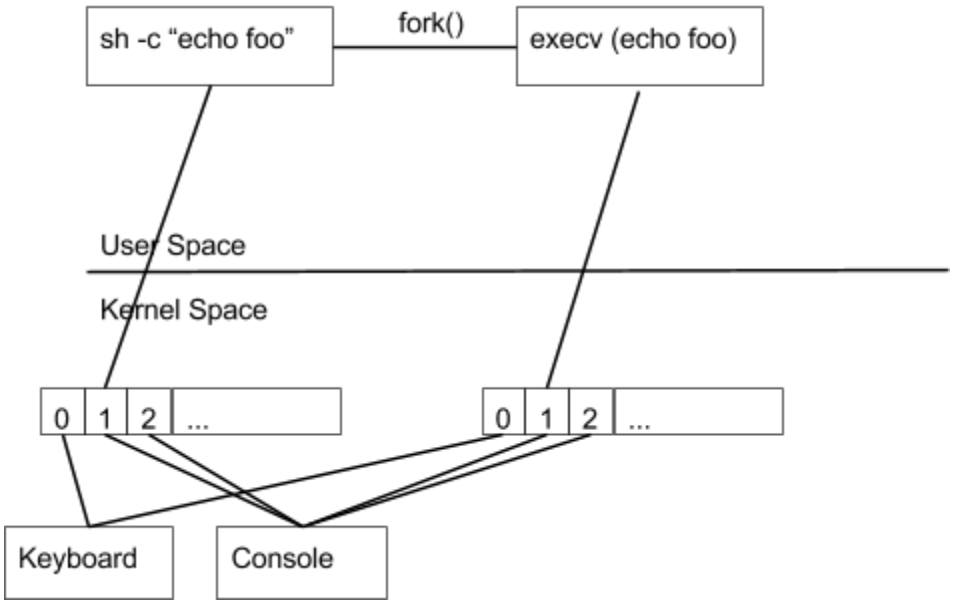
Shell: Forks, Pipes, and dupes

What happens when a process forks?

File descriptor tables: maps indexes (like 0 for standard input) to files and devices (like the console)
Running a process in a shell creates a process that is associated with a file descriptor table, has entries for every file descriptor that is open (by default, stdin, stdout, stderr for echo), 0 (stdin) is hooked up to the keyboard, 1 (stdout) and 2 (stderr) are hooked up to the screen, which is why we can't initially tell whether a program is placing output into stdout or stderr.
For example, lets represent what happens when I run "echo foo" on the command line



What happens if instead of echo foo, I run a program that runs echo foo?
sh –c “echo foo” (creates a shell that runs echo foo, looks like the same output), but under the hood it’s different. The shell can’t run that program directly (would completely replace the shell), so always runs the command in children, so forks which creates a new file descriptor table, but the contents are copied from the old parent. Then child process runs execv (replaces contents of process’s memory with “echo foo”, but exec doesn’t change file descriptor table)

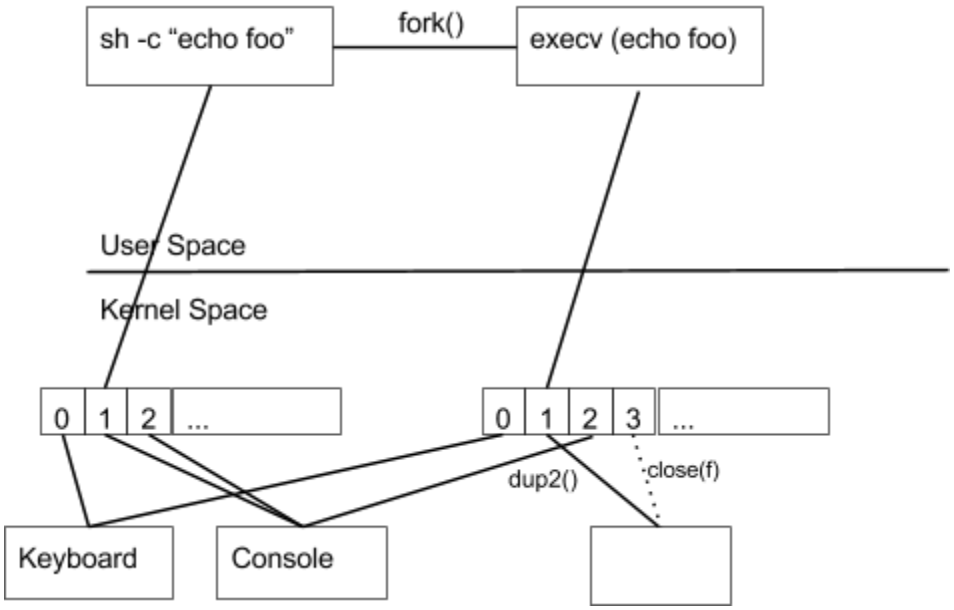


What about redirection? What if I ran echo foo > x? Takes echo foo’s output and redirects it into a file named x.
Would need to change file descriptor so that 1 points to x. Where do we do this? Can’t do this in

the parent because the parent is the shell, shell's output should be preserved! So we do it in the child.

```
p = fork();
if (p == 0) {
        // set up pipes and set up redirections
        int f = open("x", O_WRONLY | O_CREAT | O_TRUNC);
        dup2(f, STDOUT_FILENO); // erases current stdout, makes stdout a
                                //copy of what's in 3
        close(f)
        execv();
}
```

Per man pages:
int dup2(int *oldfd***, int** *newfd***);** makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary. Essentially, makes existing file descriptor copy of another file descriptor.



Why go through all of this? This is what Unix requires us to do; we need to look at Unix system calls as a fixed toolkit, which we need to be able to use to accomplish some goal.

Now let's look at a pipe:
Every program set up to assume that it should read from stdin, write to stdout, shell handles redirections of its child.

How can shell do the following:

sh -c "echo foo | wc"

Incorrect idea of how to do this: shell forks a child for echo, child creates a pipe and execs echo, shell forks another child for wc, in this 2nd child use dup2 to hook up pipe in echo to pipe in wc, then execs wc.

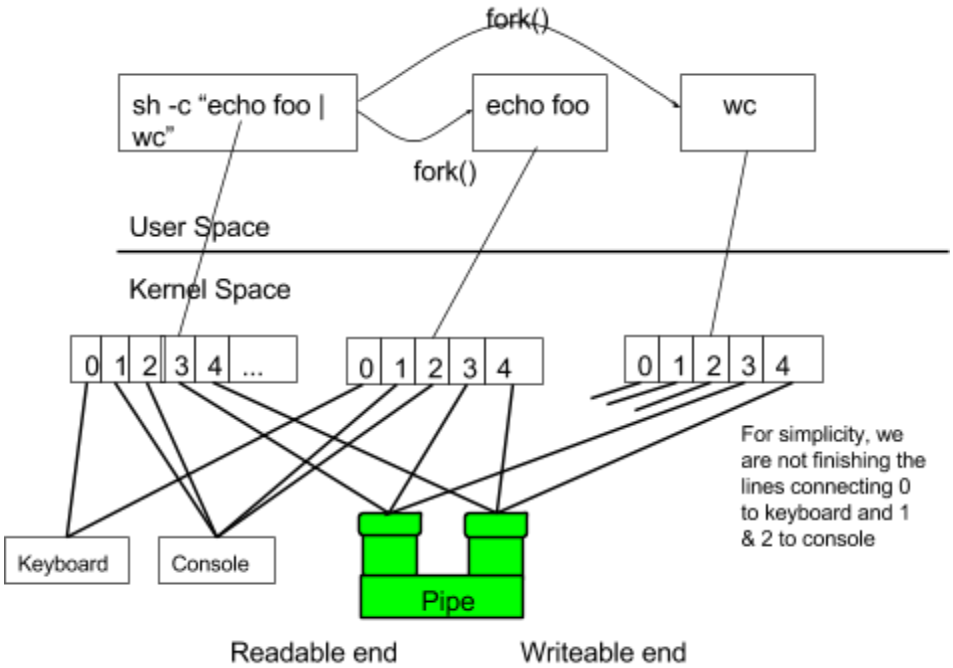
```
sh:
  fork
        pipe
        exec echo
  fork
        dup2->stdin
        exec wc
```

What's the problem? Cannot copy pipe from 1st child because of process isolation. Each processes file descriptor table is isolated, so we cannot steal a copy of a pipe from parent. This isn't going to work.

Need to create a pipe that can be shared by both processes, so create the pipe before the forks (i.e. in the shell).

Goal: output of first program is input of second one; once both children have been forked, can clean up extra file descriptors in shell.

Below is what happens when we set up the pipe in the shell and do all the forks (i.e. before we correct the file descriptors).



In first child:

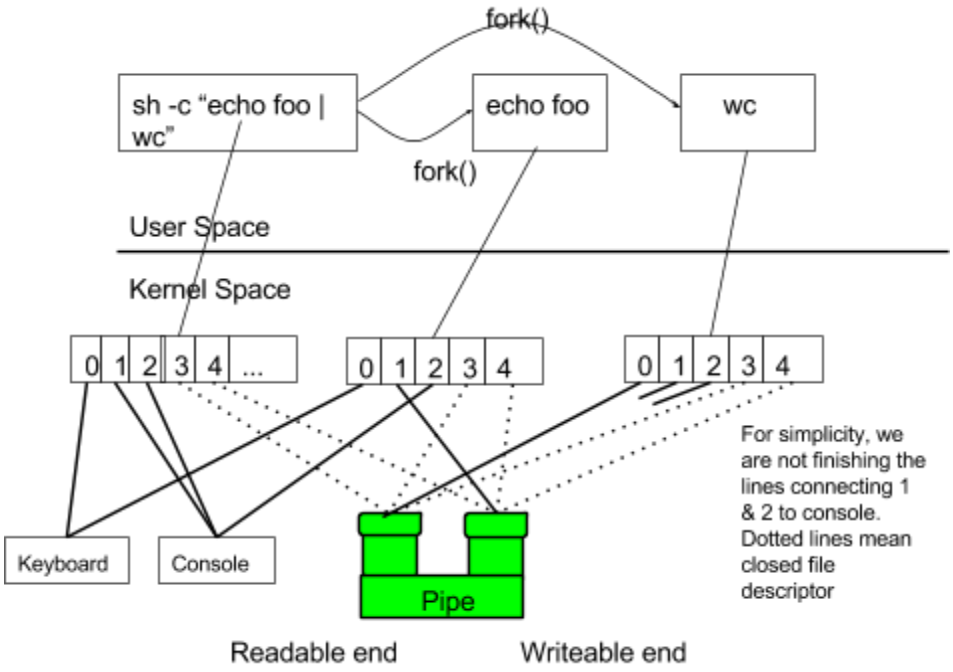
- dup2[pipefd[1], STDOUT_FILENO); // pointing to right end of pipe. We write to right/output

// end of pipe similar to how we write to stdout.

- close(pipefd[1]); // clean up -- close all other connections to pipe
- close(pipefd[0]); // clean up -- close all other connections to pipe

In second child:

- dup2 with 3 and STDIN (pointing to left end of pipe. We read from left/input end of pipe similar to how we read from stdin).
- clean up -- close all other connections to pipe



Pipe Example:

Sieve of Eratosthenes: procedure for generating prime numbers. Start with 2, cross out all of its multiples, go to the next thing that's not crossed out and cross out its multiples, then repeat.
Let's build a Sieve eratosthenes using only the following two programs:
./seq X → prints out a sequence of numbers from 1 to X.
./filtermultiples X → reads one input in command line and reprints input only if not a multiple of X (ex: ./filtermultiples 5 reprints anything not a multiple of 5 that it's given)

Attempt 1: Pipe filtermultiples of 2 to filtermultiples of 3 to ect…

```
>> seq 2 5 | (echo 2 ; ./filtermultiples 2)
 2
```

3
5

Here, parens consolidate shell calls.

```
>> seq 2 10 | sh -c "echo 2; ./filtermultiples 2" | sh -c "echo 3; ./filtermultiples 3" | cat
```

This is good until we seq up to 30. So need to add more stages to pipeline until we have Sieve of Eratosthenes.
How can we programatically create a SOE (sieve of erat.) and not do it manually?
We want each iteration to create a new call of filtermultiples.
Look at primesieve.c. This program is going to create a huge number of pipes. It creates a child running seq. seq's output goes into primeseive which is running following loop:

read number;
print number;
fork ./filtmultiples with number;

But it also needs to unhook seq from primeseive and hook it to filtermultiples call and connect output of filtmultiples call to primeseive