RECITATION 10

INFO

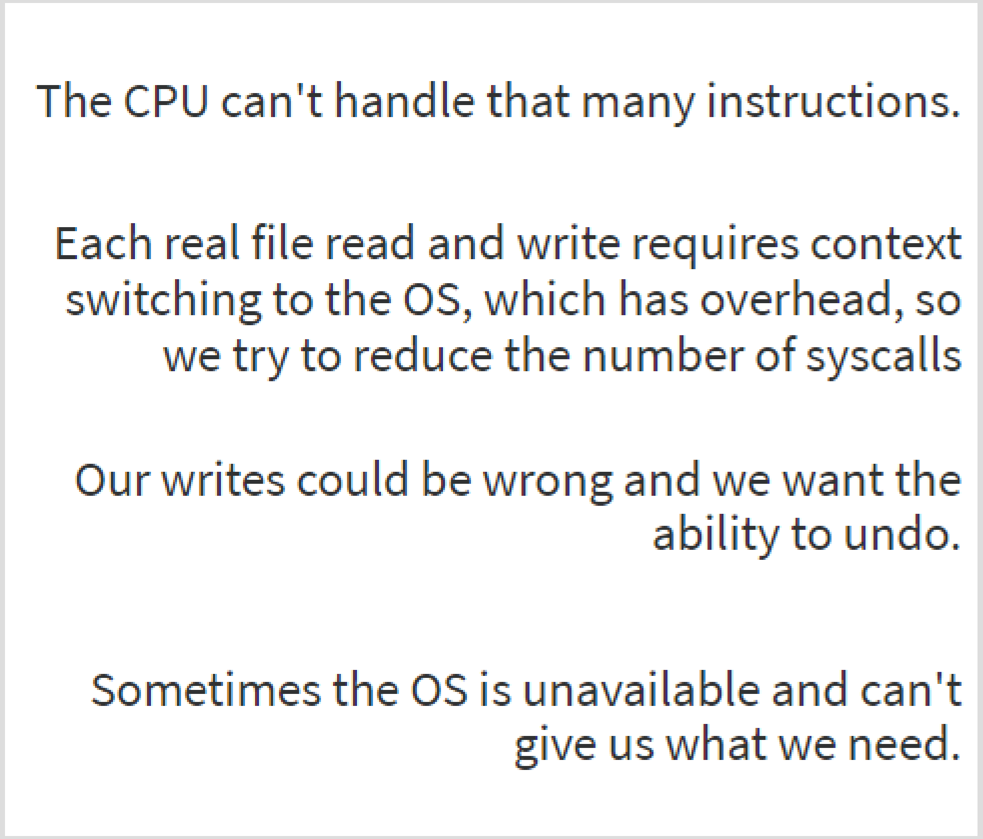
- Jon Rutkauskas
- Recitation: Tue 12-12:50
- Office Hours: Tue 11-11:50
  Thur 11-12:50
  SENSQ 5806
  (additional hours by appointment if needed)

- On discord: @jrutkauskas
- By email: jsr68@pitt.edu
- Website:
  https://github.com/jrutkauskas/spring2019-449-rec
- Ask me any questions you have!!!

WARMUP POLL

WHAT IS ONE REASON THE C STANDARD LIBRARY BUFFERS OUR FILE READS/WRITES? (AND WHAT EVEN IS A BUFFER?!)

- When we buffer… (think about it when answering)



The CPU can't handle that many instructions.

Each real file read and write requires context switching to the OS, which has overhead, so we try to reduce the number of syscalls

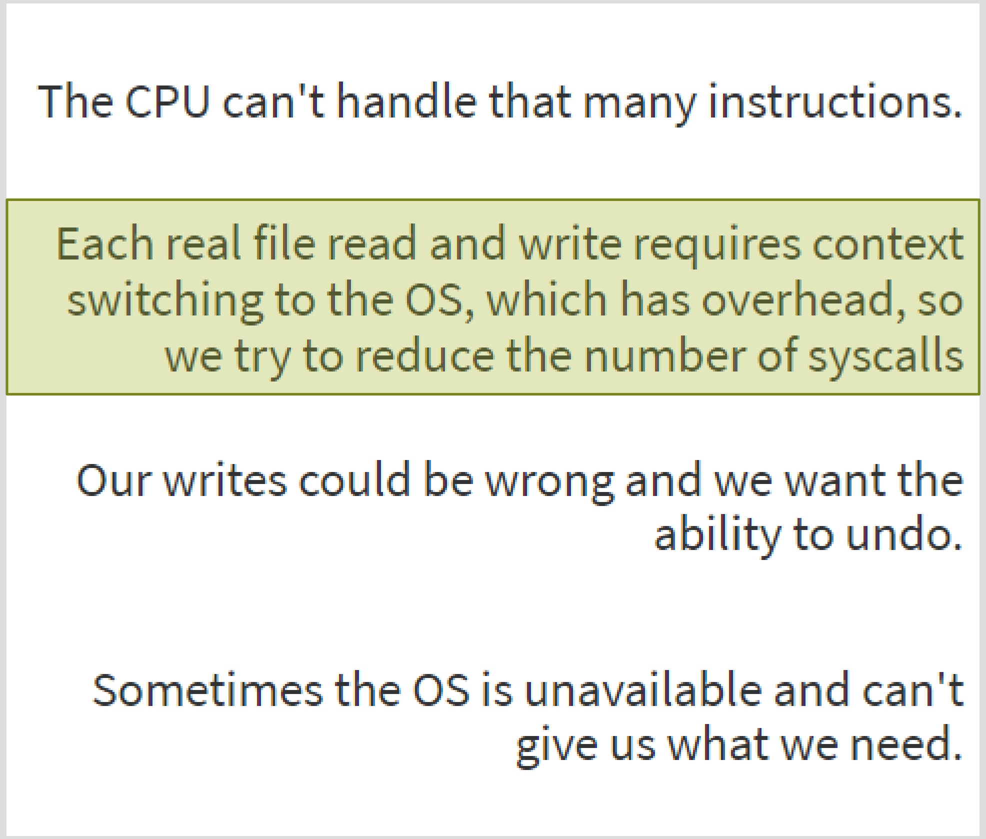
Our writes could be wrong and we want the ability to undo.

Sometimes the OS is unavailable and can't give us what we need.

WHAT IS ONE REASON THE C STANDARD LIBRARY BUFFERS OUR FILE READS/WRITES? (AND WHAT EVEN IS A BUFFER?!)

A buffer is a space in memory the C library sets aside to hold a copy of part or all of our file, so when we request to read, it gets more than we need and can give us more characters without going to the OS. When we write, it writes to the buffer instead of the file until we 'flush' so we don't have to talk to the OS as frequently

1. The CPU can handle pretty much anything we want – The CPU is incredibly fast, it's I/O like the hard drive that's really slow (~5ms)… yes, that's slow
2. Context switches are really slow. Your OS has to save all your registers, change CPU modes, and a whole lot more. This is why all syscalls are slow and you should try to be efficient with them.
3. The correctness of our writes doesn't matter, and the C library doesn't really let you 'undo'
4. The OS is always available to the process. Heck, it's only briefly giving you the privilege of running on *its* system.



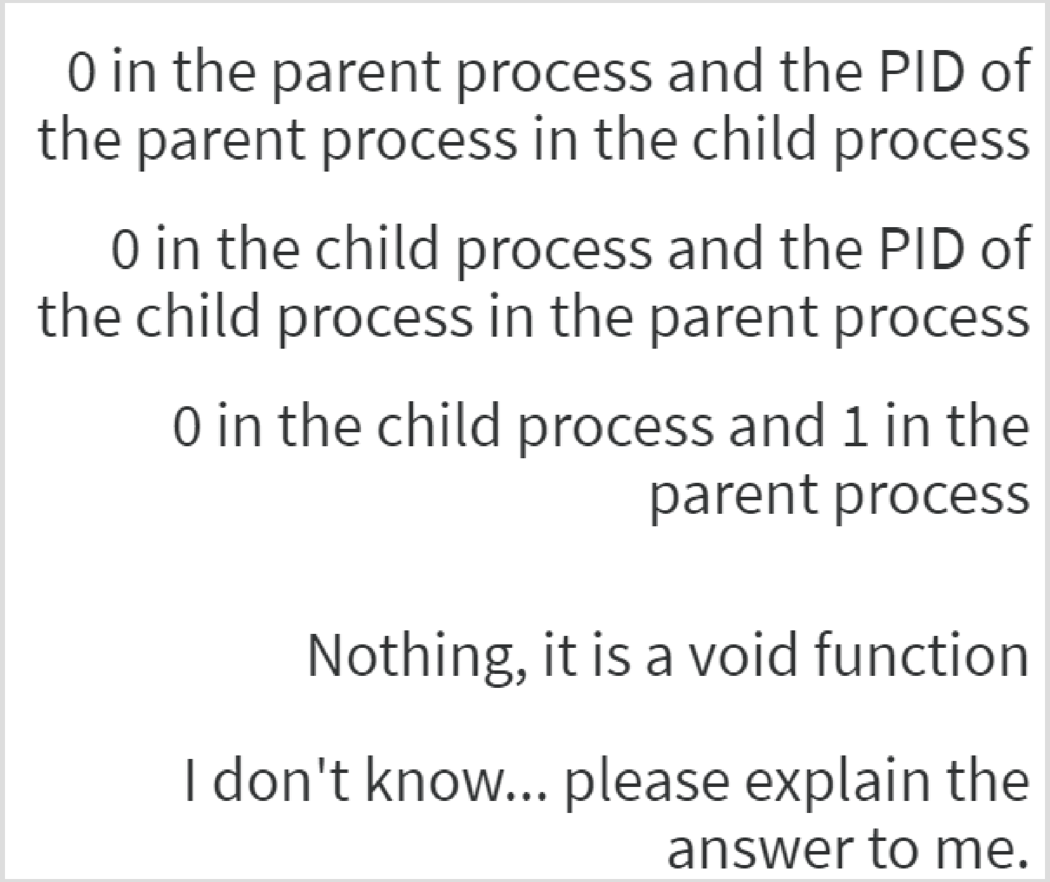
The CPU can't handle that many instructions.

[✓ Each real file read and write requires context switching to the OS, which has overhead, so we try to reduce the number of syscalls]

Our writes could be wrong and we want the ability to undo.

Sometimes the OS is unavailable and can't give us what we need.

FORK() MAKES AN EXACT COPY OF YOUR PROCESS. WHAT DOES IT RETURN?



0 in the parent process and the PID of the parent process in the child process

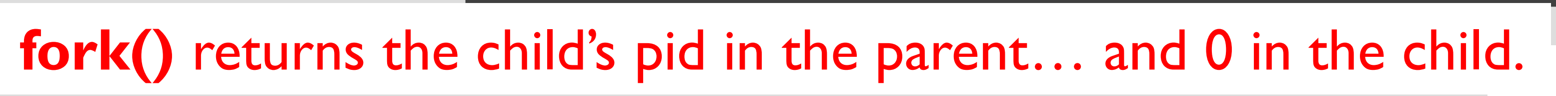
0 in the child process and the PID of the child process in the parent process

0 in the child process and 1 in the parent process

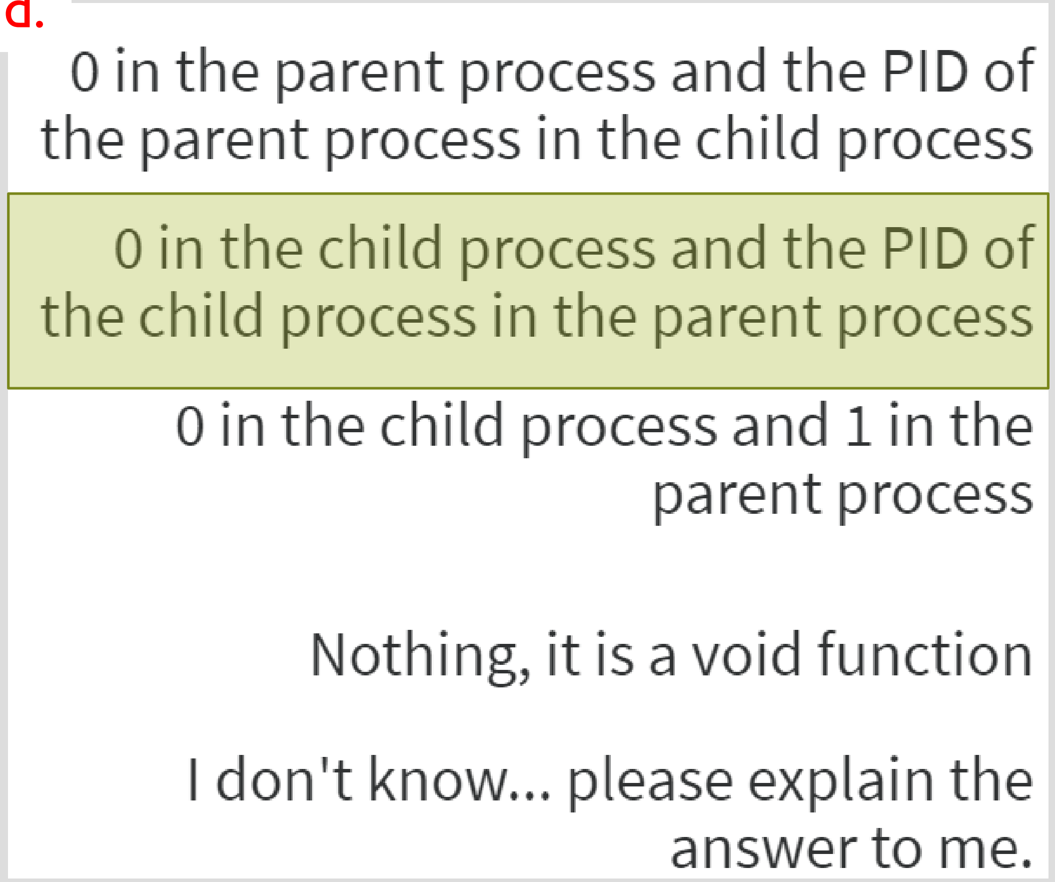
Nothing, it is a void function

I don't know... please explain the answer to me.

FORK() MAKES AN EXACT COPY OF YOUR PROCESS. WHAT DOES IT RETURN?


fork() returns the child's pid in the parent… and 0 in the child.

- ^^ From the lecture slides
- This is how we tell if we're the child or not. Usually do an if statement on it, if it's zero, do the child's work, otherwise be a parent.
- Don't mix these up or you'll be very confused on what your program is doing.
- Also, remember that after doing fork(), the child process is an exact clone but runs independently (might get more time on the CPU than the parent, or finish sooner)


0 in the parent process and the PID of the parent process in the child process

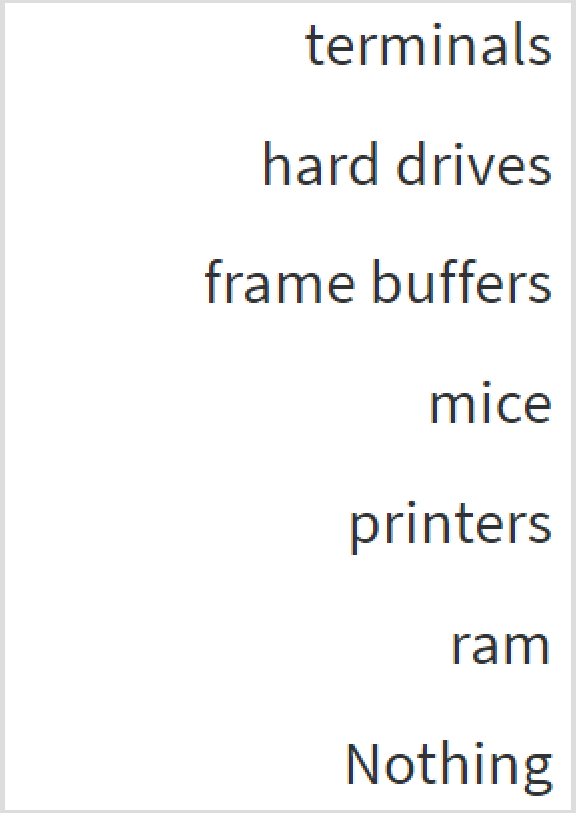
[✓ 0 in the child process and the PID of the child process in the parent process]

0 in the child process and 1 in the parent process

Nothing, it is a void function

I don't know... please explain the answer to me.

WHICH OF THE FOLLOWING IS NOT ABLE ACCESSED THROUGH FILES IN POSIX, GENERALLY?



terminals

hard drives

frame buffers

mice

printers

ram

Nothing

WHICH OF THE FOLLOWING IS NOT ABLE ACCESSED THROUGH FILES IN POSIX, GENERALLY?

- **It's all files**
- Like, everything is files
- So much files
- Just ls on /dev, and look at all those "files"
- (Technically, even 'nothing' is a file, /dev/null)
- Another fun file: /dev/urandom

terminals

hard drives

frame buffers

mice

printers

ram

[✓ Nothing]

PROJECT 4

- You know the shell, you use it all the time… it's bash!
- Haha, its job's not that hard, right?
- You could do that in your sleep, right?
- Well, now you will, rhetorical 449 student!
- Project 4 – The Shell Project
- (Coming soon because I prepped these slides thinking the project would be released on time)

PROJECT 4

- Don’t worry, it’s not as nearly complicated as actual bash
- Just a basic little interactive shell
- The shell segments into very easy functions, so this project breaks down into nice chunks
- Really, most of this project is just learning how to read man pages and work with functions
- Lots of OS interactions – Good practice for 1550.

PROJECT 4

- You'll make functions for your interactivity (start with the prompt and reading in the user input)
- Then, you can write functions to parse the input and call other functions to handle things like running programs, changing directories, redirecting output, etc.
  - Remember, Top-Down design here
- Then, you can read the man pages for the functions you need to call, and it's pretty straightforward from there, just call them as requested and HANDLE ERRORS (of which there will be many)

DON'T FORKBOMB THOTH

- Don't Forkbomb Thoth
- Don't Forkbomb Thoth
- Don't Forkbomb Thoth
- Don't Forkbomb Thoth
- Don't Forkbomb Thoth
- Don't Forkbomb Thoth
- Don't Forkbomb Thoth
- Everyone will be upset… even (and especially) 1550 students

DON’T FORKBOMB THOTH

- EVERY TIME YOU LOGIN
  - Run **`ulimit –u 40`**
  - This will limit you to only 40 processes (currently it’s set to something like 512)
  - That way, if you forkbomb, you’ll hit your 40 process max you’ve set and it will stop.
  - Then, all you have to do is disconnect from ssh, open it up again, log back in, and run the command “**`killall -9 myshell`**” (or whatever you called your executable)
  - This will kill every shell process you made, thus stopping the forkbomb
  - If you don’t do this, the system resources will get bogged down from your processes and EVERYONE will be slowed down to an unusable crawl and there will be nothing you can do except ask Jarrett to kill the processes for you (since you can’t get a bash process or killall process, etc)
  - So PLEASE, use this command. Please. And yes, you have to run it every time you login.

EMPHASIS

- Repeated for emphasis:
- use the following command:

```
ulimit -u 40
```

- Every time you login
- If you accidentally forkbomb, re-log in. Then you can run:

```
killall -9 <the name of your shell executable>
```

- and that will kill any of your shell processes.

PROJECT 4

- Message me if you have questions!
- Even if I'm offline I'll get a notification on my phone and will get back to you.

LAB 7

- It's out! – Basically, it gets you started on part of the project ;-)
- We'll be trying to use fork and execvp to run programs from our programs!
  - Actually, that part is done for you!
  - Follow along with the comments in the code.
  - You just have to do some error handling!
  - And get some good practice with man pages!
  - In fact, let's get some man practice right now!

```
[thoth ~/private/cs449/lab7]: man perror
```

perror(3) - Linux man page

Name

perror - print a system error message

Synopsis

#include <stdio.h>
void perror(const char ****s*);**
#include <errno.h>
const char ****sys_errlist*[];**
int *sys_nerr***;**
int *errno***;**
Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):
sys_errlist, *sys_nerr*: _BSD_SOURCE

Description

The routine **perror**() produces a message on the standard error output, describing the last error encountered during a call to a system or library function. First (if *s* is not NULL and **s* is not a null byte ('\0')) the argument string *s* is printed, followed by a colon and a blank. Then the message and a new-line.
To be of most use, the argument string should include the name of the function that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when successful calls are made.
The global error list *sys_errlist*[] indexed by *errno* can be used to obtain the error message without the newline. The largest message number provided in the table is *sys_nerr*-1. Be careful when directly accessing this list because new error values may not have been added to *sys_errlist*[]. The use of *sys_errlist*[] is nowadays deprecated.
When a system call fails, it usually returns -1 and sets the variable *errno* to a value describing what went wrong. (These values can be found in <*errno.h*>.) Many library functions do likewise. The function **perror**() serves to translate this error code into human-readable form. Note that *errno* is undefined after a successful library call: this call may well change this variable, even though it succeeds, for example because it internally used some other library function that failed. Thus, if a failing call is not immediately followed by a call to **perror**(), the value of *errno* should be saved.

Conforming To

The function **perror**() and the external *errno* (see **errno**(3)) conform to C89, C99, 4.3BSD, POSIX.1-2001. The externals *sys_nerr* and *sys_errlist* conform to BSD.

Notes

The externals *sys_nerr* and *sys_errlist* are defined by glibc, but in <*stdio.h*>.

See Also

err(3), **errno**(3), **error**(3), **strerror**(3)

Referenced By

explain(3), **explain_lca2010**(1), **fmtmsg**(3), **genders_errnum**(3), **lam-helpfile**(5),**nodeupdown_errnum**(3), **psignal**(3), **rmt**(8)

perror(3) - Linux man page

Name
perror - print a system error message
Synopsis
#include <stdio.h>
void perror(const char *_s_);
#include <errno.h>
const char **sys_errlist*[];
int *sys_nerr*;
int *errno*;
Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):
sys_errlist, *sys_nerr*: _BSD_SOURCE

Description
The routine **perror**() produces a message on the standard error output, describing the last error encountered during a call to a system or library function. First (if *s* is not NULL and **s* is not a null byte ('\0')) the argument string *s* is printed, followed by a colon and a blank. Then the message and a new-line. To be of most use, the argument string should include the name of the function that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when successful calls are made.
The global error list *sys_errlist*[] indexed by *errno* can be used to obtain the error message without the newline. The largest message number provided in the table is *sys_nerr*-1. Be careful when directly accessing this list because new error values may not have been added to *sys_errlist*[]. The use of *sys_errlist*[] is nowadays deprecated.
When a system call fails, it usually returns -1 and sets the variable *errno* to a value describing what went wrong. (These values can be found in <*errno.h*>.) Many library functions do likewise. The function **perror**() serves to translate this error code into human-readable form. Note that *errno* is undefined after a successful library call: this call may well change this variable, even though it succeeds, for example because it internally used some other library function that failed. Thus, if a failing call is not immediately followed by a call to **perror**(), the value of *errno* should be saved.
Conforming To
The function **perror**() and the external *errno* (see **errno**(3)) conform to C89, C99, 4.3BSD, POSIX.1-2001. The externals *sys_nerr* and *sys_errlist* conform to BSD.
Notes
The externals *sys_nerr* and *sys_errlist* are defined by glibc, but in <*stdio.h*>.
See Also
err(3), **errno**(3), **error**(3), **strerror**(3)
Referenced By
explain(3), **explain_lca2010**(1), **fmtmsg**(3), **genders_errnum**(3), **lam-helpfile**(5),**nodeupdown_errnum**(3), **psignal**(3), **rmt**(8)

FUNCTIONS YOU MAY BE USING

- `perror`
- `exit`
- `execvp`
- `signal`
- `waitpid`
- `WIFEXITED(status)`
- And more! (view the man page for waitpid to read about some more functions that are related to it)