Recitation 1

Hope everyone is settling in well!

Administrivia

- PennVector (2.5% of cumulative grade)
  - Due F 09/04 on Gradescope (really W 09/09)
- PennShell (10% of cumulative grade)
  - Milestone 0 due F 09/11
  - Milestone 1 due F 09/18
  - Final version due W 09/23
  - Find a partner!!! Use Ed, google form, etc. (maybe talk to the person next to you...)
- Pre-Semester Survey (5% of cumulative grade)
  - Due W 09/09 on Canvas

Agenda

- Tools
- C memory
- `wait` and `waitpid`
- Signals
- Pennshell demo

Man Pages

The Man Pages

1.1 The Man Pages

Let's try looking up a couple of different man pages for different functions in C.

- Type man read in the terminal. Do you see what you expect?

  No

- What do you need to type in order to see the correct man page for the C function for read?

  man 2 read

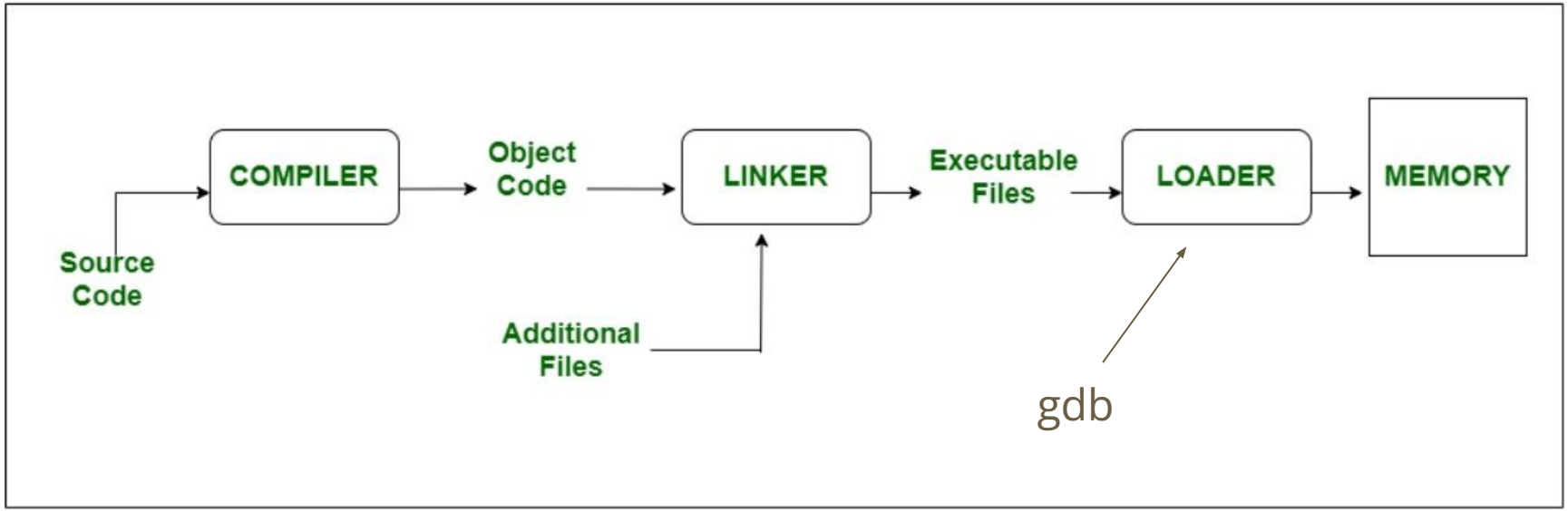
- What about man printf? Is it in the same section as read? Why or why not?

  man 3 printf; read is POSIX, printf is C stdlib

GDB

How do Programs Run? (gcc)

- gcc (GNU Compiler Collection)
  - We use makefiles, but these rely on gcc!
  - For more: https://web.stanford.edu/class/archive/cs/cs107/cs107.1202/resources/gcc



What is GDB?

GNU (GNU's Not Unix) Project Debugger

- You can can examine the program as it is executing!
- Supports a variety of languages
  - (e.g. asm, C, C++, D, Fortran, Go, Rust, etc.)
- ***Your best friend for this course!***

"But can I use print statements?"

- Using print statement can be tedious.
- Requires the use of many to find where the bug is.
- Unable to examine the state of the program after it crashes.
- `printf("here\n");` can only take you so far.

Using GDB

Use `-g` or `-g3` flag in Makefile for compiling (the provided Makefile has this)

Type in your shell:

a) gdb pokemon_buggy

b) <enter>, then run

start	Start from beginning and stop there
run	Start and run program from beginning
continue	Run until program exits*
step	Run until next line*
bt	Shows call stack
b [fname:]function b [fname:]linenum	Sets breakpoint at beginning of function or at line
print var	Prints var

* = or until next breakpoint

Valgrind

Valgrind

- Your handy debugging tool for memory mismanagement
- It runs your program, and looks for any memory errors during execution (e.g.: memory leaks)
- It will only catch errors it encounters in runtime! Pay attention to **code coverage** - ensure all* lines of code are run in a valgrind session

*or at the very least, the most critical lines

Valgrind Errors

- Memory leaks: memory that hasn't been freed before the program exits
- Invalid read/write: accessing unallocated (or deallocated) memory
- Uninitialized bytes: Using memory that was allocated but never had any values put into them

How to Run Valgrind

- Can run in terminal: **`valgrind ./program <program arguments>`**
- Useful valgrind arguments (put between `valgrind` and `./program`)
  - **`--trace-children=<yes|no>`** (default: no)
  - **`--track-origins=<yes|no>`** (default: no)
  - **`--leak-check=<no|summary|yes|full>`** (default: summary)
    - `full` option gives you the most info

Valgrind Demo!

C Memory Layout

C Memory Layout

2.1 Where Does This Live?

For this question, we will review the segments of memory in a typical C program. Let's go ahead and get up to speed.
Consider the following code:

```
#include <stdio.h>

int g = 7;

int main(void) {
    char arr[] = "hello";
    char *ptr = "world";

    printf("%c %c %d\n", arr[0], ptr[0], g);
    return 0;
}
```

Question: For each item below, identify where it exists in the process address space layout (stack, heap, .data/.bss, or read-only text/rodata), and briefly justify.

- The variable arr
- The characters that make up the string stored in arr (i.e., the memory backing "hello")
- The variable ptr
- The characters that ptr points to (i.e., the memory backing "world")
- The variable g

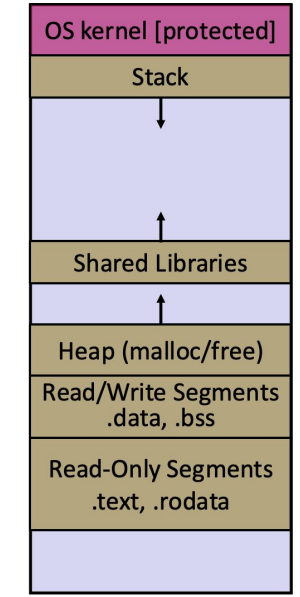
arr: Stack

Memory holding "hello": Stack, initialized from ro

ptr: Stack

Memory holding "world": rodata

g: data (r/w)



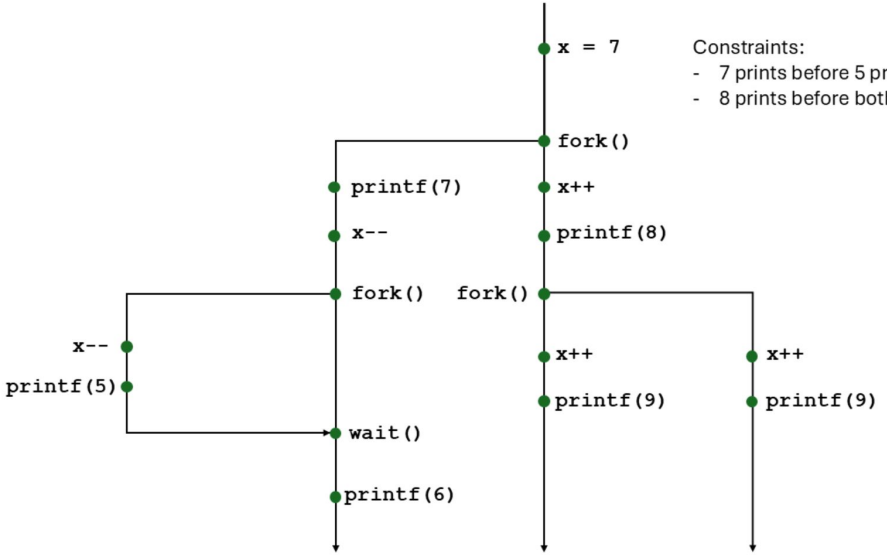
fork

fork and wait

3.1 Process Graph and Outputs

Consider the following code using fork and wait. Draw out a process graph and write four potential outputs (i.e., order of things printed) for the code. What constraints are guaranteed to hold for all the potential outputs (i.e., what about the ordering of certain values must always be true)?

```
int x = 7;
if (fork()) {
    x++;
    printf("%d\n", x);
    fork();
    x++;
    printf("%d\n", x);
} else {
    printf("%d\n", x);
    x--;
    if (!fork()) {
        x--;
        printf("%d\n", x);
    } else {
        wait(NULL);
       printf("%d\n", x);
    }
}
```



Constraints:
- 7 prints before 5 prints before 6
- 8 prints before both 9s

wait, waitpid, and Blocking

wait, waitpid, and Blocking

- `int wait(int* status)`: Parent waits for any child to finish - blocks while waiting
- A process can only wait on its child; no sibling or grandchild waiting allowed!
- `int waitpid(int pid, int* status, int options)` is more expressive
  - `pid` specifies the child; `-1` if waiting for any child
  - `options` specifies the type of waiting
    - Blocking wait by default
    - Nonblocking wait with `WNOHANG`
    - Wait for stopped/continued children with `WUNTRACED/WCONTINUED`
- `status` indicates the change via macros
  - `WIFEXITED(status)`, `WIFSIGNALED(...)`, `WIFSTOPPED(...)`, `WIFCONTINUED(...)`

wait, waitpid, and Blocking

3.1 Predict Before Running

Consider the following program:

```
int main() {
  pid_t pid = fork();
  if (pid == 0) {
    sleep(5);
    printf("Child done\n");
    return 0;
  }
  printf("Parent waiting...\n");
  wait(NULL);    //    wait for any child to finish
  printf("Parent done\n");
}
```

Question:

- Which line(s) could appear first? Which can never appear first?

- Suppose we remove the wait call in the parent. What happens to the child?

"Child done" and "Parent waiting..." can appear first.

"Parent done" can never appear first.

The child becomes an orphan (parent returned from main before waiting on the child, so init adopts child).

wait, waitpid, and Blocking

3.2 Busy Waiting vs Blocking

Now consider this version:

```
int status;
while (waitpid(-1, &status, WNOHANG) == 0) {
  printf("Still waiting...\n");
}
printf("Done waiting\n");
```

Question:

• What does it mean when waitpid returns 0 in this loop?

• You realize that the child you are waiting for might never return. How can you terminate it?

waitpid returning 0 means the child has not terminated.

Send a terminating signal to the child (open a new terminal and type kill -[signal] [child_pid]).

Signals

Signals

- Software interrupt - a notification sent to a process
- An example of inter-process communication
  - We will see another example of IPC soon...
- From the OS: some sort of exception has occurred
- From another process: used to coordinate activity or as notification
- Examples: SIGINT (interrupt, Ctrl-C), SIGTSTP (terminal stop, Ctrl-Z), SIGKILL (force kill), SIGCHLD (child changed state)

Default dispositions + modifying that behavior

- Signal dispositions = what they do when the signal is received.
  - Some stop the process, e.g. SIGTSTP
  - Some are ignored by default, e.g. SIGCHLD
  - Some cause a core dump, e.g. SIGSEGV (segfault)
- What if we don't like the default behavior?
- Sigaction allows us to override most signals (not SIGKILL, SIGSTOP) with SIG_IGN, SIG_DFL or self defined signal handlers.

What if we don’t want a signal?

- Could ignore …
- What if we want it later?
  - critical section (pause signal for now so not malformed!)?
  - Reaping without waiting (penn-shell extra challenge)?
- Block and set signal to pending with sigprocmask
  - How do we reset? sigprocmask again with original mask

Activity: Responses to Signals

In each scenario, determine whether the signal was blocked, ignored, used default handling, or used a handler function (non-default)

1. In penn-shell, I hit Ctrl+Z. In response, penn-shell stopped, and now I'm back in the host shell.

Activity: Responses to Signals

In each scenario, determine whether the signal was blocked, ignored, used default handling, or used a handler function (non-default)

1. In penn-shell, I hit Ctrl+Z. In response, penn-shell stopped, and now I'm back in the host shell.
   Default Action (Ctrl+Z = SIGTSTP)
2. In penn-shell, I hit Ctrl+Z. In response, penn-shell reprompted.

Activity: Responses to Signals

In each scenario, determine whether the signal was blocked, ignored, used default handling, or used a handler function (non-default)

1. In penn-shell, I hit Ctrl+Z. In response, penn-shell stopped, and now I'm back in the host shell. **Default Action (Ctrl+Z = SIGTSTP)**
2. In penn-shell, I hit Ctrl+Z. In response, penn-shell reprompted. **Signal Handler Function**
3. A process received a signal, and its signal handler called vec_push. While in the middle of executing vec_push, it received the same signal again. The program continues the handler, and once it finishes, the signal handler ran again.

Activity: Responses to Signals

In each scenario, determine whether the signal was blocked, ignored, used default handling, or used a handler function (non-default)

1. In penn-shell, I hit Ctrl+Z. In response, penn-shell stopped, and now I'm back in the host shell. **Default Action (Ctrl+Z = SIGTSTP)**
2. In penn-shell, I hit Ctrl+Z. In response, penn-shell reprompted. **Signal Handler Function**
3. A process received a signal, and its signal handler called vec_push. While in the middle of executing vec_push, it received the same signal again. The program continues the handler, and once it finishes, the signal handler ran again. **Signal Blocked, then Unblocked**

Signals

4.1 Not Handling Signals in a Signal Handler

Signal handlers can interrupt one another; this can be dangerous. To fix this, you can set the `sigset_t sa_mask` field of a `sigaction`, which automatically applies when the handler is running.

```
struct sigaction sa_sigusr1 = {.sa_handler = handler_a}
struct sigaction sa_sigusr2 = {.sa_handler = handler_b}

// Fill in the (4) lines that make this safe.
sigemptyset(&sa_sigusr1.sa_mask);
sigaddset(&sa_sigusr1.sa_mask, SIGUSR2);
sigemptyset(&sa_sigusr2.sa_mask);
sigaddset(&sa_sigusr2.sa_mask, SIGUSR1);

sigaction(SIGUSR1, &sa_sigusr1, NULL);
sigaction(SIGUSR2, &sa_sigusr2, NULL);
```

Signals

4.2 Reasoning About Atomicity

Consider the following program:

```
sig_atomic_t received = 0; // set to 1 inside a SIGINT handler
sigset_t block;
sigset_t suspend_mask;

sigemptyset(&block);
sigemptyset(&suspend_mask);
sigaddset(&block, SIGINT);

sigprocmask(SIG_BLOCK, &block, NULL);

while (!received) {
  sigsuspend(&suspend_mask);
}
```

Questions:

- What is the signal mask of the process:
  - immediately before calling sigsuspend?
  - while inside sigsuspend?
  - immediately after sigsuspend returns?

- Suppose SIGINT arrives **after** the while condition is checked but **before** sigsuspend is called. Why does this pattern still work correctly?

Before sigsuspend: {SIGINT}

While inside suspend: {}

After sigsuspend: {SIGINT}

Before sigsuspend, SIGINT is blocked and it becomes pending. sigsuspend atomically unblocks SIGINT, so SIGINT is handled and sets received = 1.

Food for thought: how should sigsuspend deal with signals which have SA_RESTART enabled?

Penn Shell Demo

Anatomy of a Shell

A shell is a prompt → read → execute loop:

1. Print prompt, reads user input, and parses
2. Create child process (fork)
3. Make child process run user input (execvp)
4. Wait for child process to finish (waitpid)
5. Repeat

There are (many!) more subtleties, but this is the core idea of a shell.

File Descriptors/File Descriptor Table

Process-Level File Descriptor Tables (FDT)

- Each process has its own file descriptor table managed by the OS
- A file descriptor (type int) is an index into a processes FD table
- Children made via fork inherit an identical FD table from their parent
    - Those files are not closed nor are they modified in any way. They are the exact same files referred to by the parent.
- Files opened exclusively in a process after a fork do not modify the FD table of another process, child or parent.

open(), close(), read(), write()

```
int open(const char* pathname, int flags, /* mode_t perm */ )
```

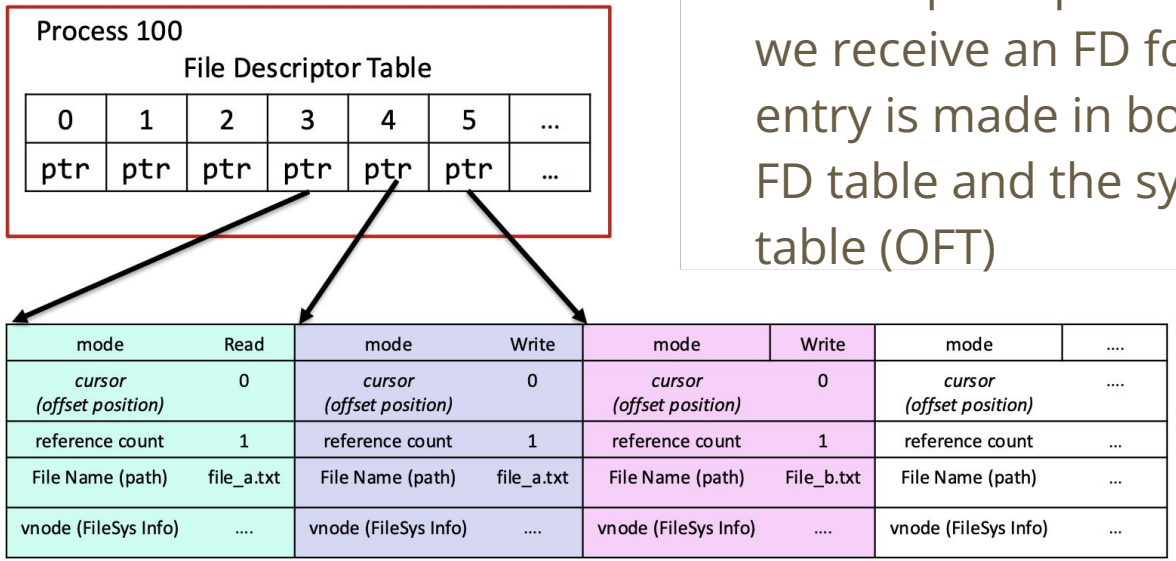
```
int close(int fd)
```

```
ssize_t read(int fd, void *buf, size_t count);
```

```
ssize_t write(int fd, void *buf, size_t count);
```

System-Wide Open File Table (OFT)

- Each entry in the process-level FD Table has a pointer to an entry in the system-wide open file table maintained by the kernel!



- As we open up more files, using open(), we receive an FD for that file AND an entry is made in both the process-level FD table and the system-wide open file table (OFT)

File Descriptor Table for Process 1

EX 1

0	1	2	3
ptr	ptr	ptr	ptr

fork()

mode	Read	mode	Write	mode	Write	mode	Read
cursor (offset position)	0	*cursor (offset position)*	0	*cursor (offset position)*	0	*cursor (offset position)*	0
reference count	1	reference count	1	reference count	1	reference count	1
File Name (path)	/dev/tty	File Name (path)	/dev/tty	File Name (path)	/dev/tty	File Name (path)	hi.txt
vnode (FileSys Info)	....	vnode (FileSys Info)	....	vnode (FileSys Info)	....	vnode (FileSys Info)	...

hi.txt

File Descriptor Table for Process 1

0	1	2	3
ptr	ptr	ptr	ptr

fork()

File Descriptor Table for Process 2

0	1	2	3
ptr	ptr	ptr	ptr

mode	Read	mode	Write	mode	Write	mode	Read
cursor (offset position)	0	*cursor (offset position)*	0	*cursor (offset position)*	0	*cursor (offset position)*	0
reference count	2	reference count	2	reference count	2	reference count	2
File Name (path)	/dev/tty	File Name (path)	/dev/tty	File Name (path)	/dev/tty	File Name (path)	hi.txt
vnode (FileSys Info)	....	vnode (FileSys Info)	....	vnode (FileSys Info)	....	vnode (FileSys Info)	...

hi.txt

Note:

In this class, the 2 main ways multiple entries from process-level FDTs point to the same entry in the system-wide Open File Table (OFT) are

1. Fork()
2. Dup2()

Redirection

Dup2

- We can manipulate the File Table so that a FD Table entry is associated with another file.
- **int dup2(int oldfd, int newfd);**
  - The file descriptor newfd is adjusted so that it now refers to the same open file pointed to by oldfd.
  - (newfd file is closed silently)

Pipes

Pipelines

What is a pipe?

- Kernel buffer with two file descriptors - one for each end
- Push from/pull from buffer from write and read end respectively

Pipelined functions: i.e. cat log | grep brains | wc -l

- What does this do? Output log → grep for lines that start with “brains” → count number of such matching lines

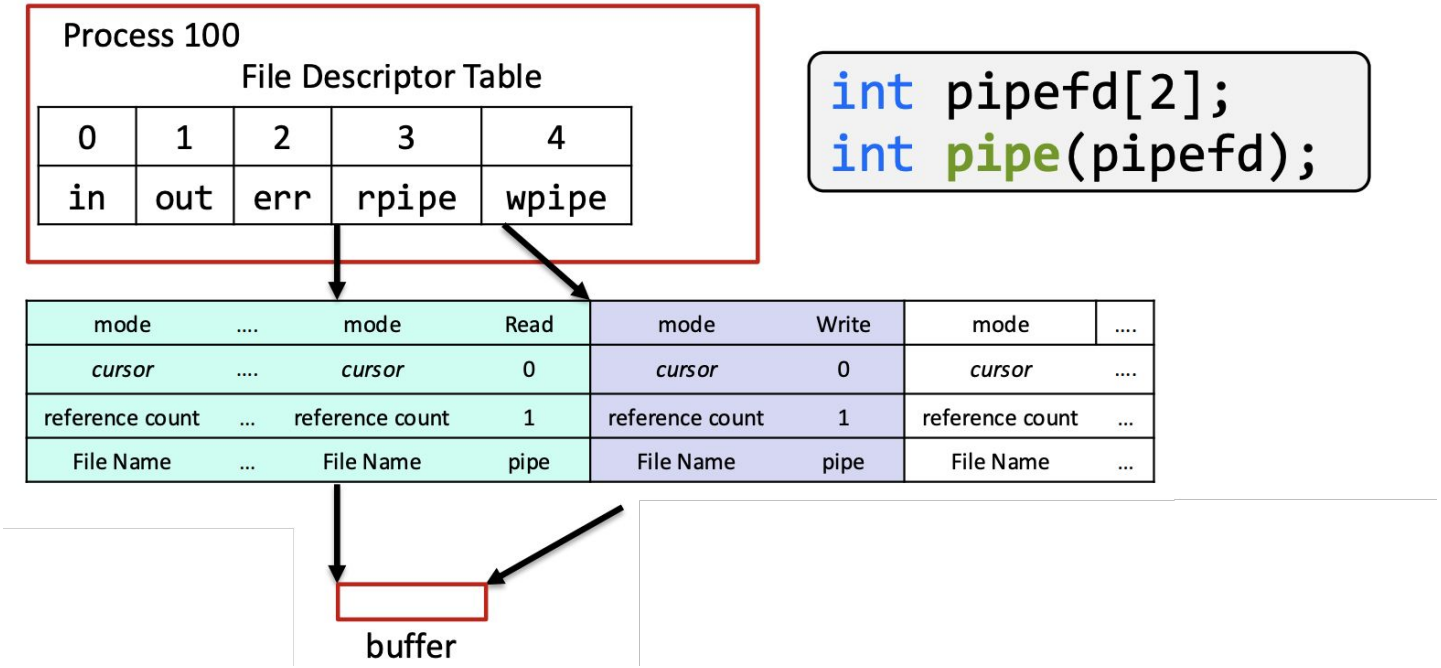
Pipelines

But how does this happen?

For an N-stage pipeline:

1) Fork N children, create N-1 pipes
2) In each child i:
    a) Dup2 pipes from input/output to STDIN_FILENO and STDOUT_FILENO
    b) Close pipes
    c) Execvp command
3) Cleanup in parent

Pipe Visualization



Pipelines Check-In

- Consider the following pipelined command:

```
# sleep 1 | sleep 20 | sleep 100
```

  How long does it take to finish?

Pipelines Check-In

- Consider the following pipelined command:
  ```
  # sleep 1 | sleep 20 | sleep 100
  ```

  How long does it take to finish?
  100 seconds

  Why?

Pipelines Check-In

- Consider the following pipelined command:
  ```
  # sleep 1 | sleep 20 | sleep 100
  ```

  How long does it take to finish?
  100 seconds

  Why?
  Pipelined processes run in parallel