CMSC 412
PROJECT 0: PIPES

Minimum Requirements: None

TEST DISTRIBUTION

Public tests – 5 tests | 22 points

Release tests – 1 test | 5 points

Secret tests – 5 tests | 25 points

SYSTEM CALLS

- A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
- Calling function(pipe, read, write) in an user executable(pipe-p1.c) will end up automatically calling its corresponding system call (Sys_Pipe, Sys_Read, Sys_Write). Please note that binding of all system calls is in src/libc/fileio.c
- Flow of Pipe Create:
  Pipe-p1.c→fileio.c→syscall.c → pipe.c

PIPE SYSTEM CALL

- A pipe is a system call that creates a unidirectional **communication link between two file descriptors**.
- A file descriptor is a **number that uniquely identifies an open file in a computer's operating system**.
- int Pipe(int *fd_read, int *fd_write) takes two arguments: **each is a pointer to an integer location**.
- When Pipe returns successfully, it would have created a pipe and filled the two location with file descriptors(integers), one pointing to the reading end of the pipe and the other to the writing end of the pipe

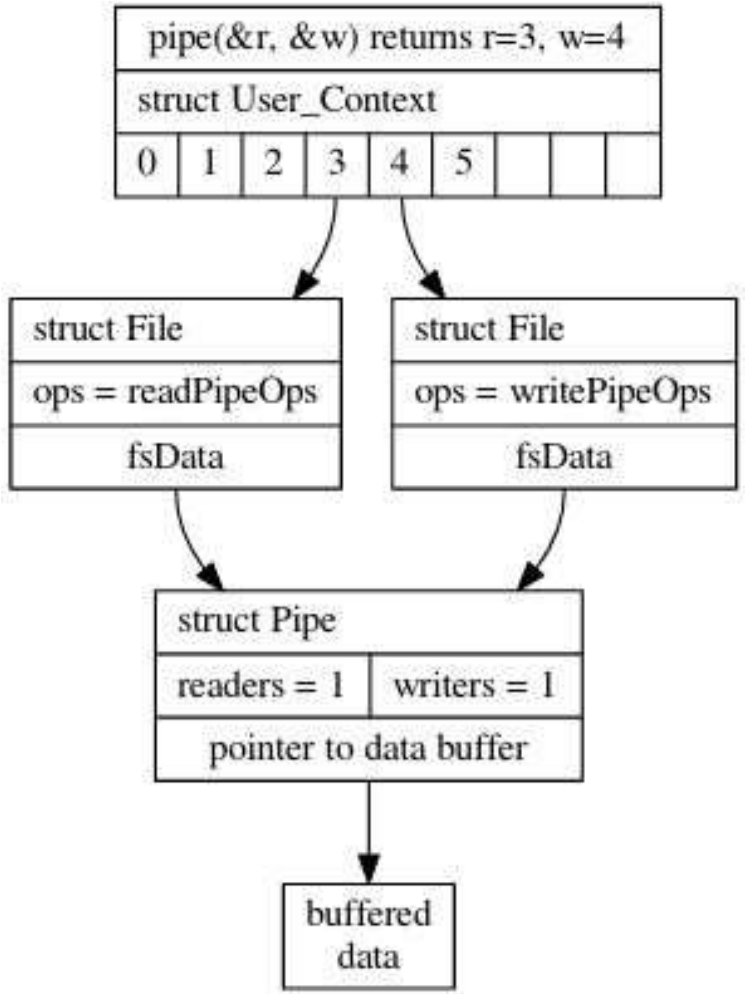
STRUCTS

- Struct File in vfs.h
- Struct FileOps in vfs.h
- Struct pipe: you need to create this one

PIPE_CREATE()

REFERRED TO AS PIPE() IN PROJECT SPEC)

- Two File double pointers (READ_FILE AND, WRITE_FILE) have been passed to populate the file struct.
- Create new struct File instance using Malloc() or Allocate_File()
- Initialize necessary fields in the file struct.
- There are File_Ops defined in the pipe.c file.
- Need to have your own pipe struct to hold data and other variables of importance (as per your judgement).
  - The data buffer could be a fixed 32K or dynamically allocated buffer.
- Use fsData(void*) in file to point to the instance of your pipe struct
- Check for appropriate error conditions wherever necessary.
- Return 0 if successful



SYS_PIPE()

- This is what is called when Pipe() command is executed in the test files (user mode)
- Create the pipe (call Pipe_Create()).
- Add files to the descriptor table.
- Look at add_file_to_descriptor_table function.
- Use Copy_To_User(ulong_t destInUser, const void *srcInKernel, ulong_t bufSize) to copy the file descriptors to the user addresses stored in the state registers (refer to geekos slides on how to use).
- Remember the addresses in the state registers are memory addresses in user space, the code you are writing is in kernel space.
- Return 0 if successful, remember to check for error conditions through out this function.

TESTING

- At this point, your code should be able to create a pipe.
- Try to run pipe-p1 and it should pass the first assertion without any error.

PIPE_READ()

(REFERRED TO AS READ() IN PROJECT SPEC)

- Goal: Reads data from the pipe into the buffer
- Inputs: num_bytes you have to read from the pipe, a buffer to copy data into and a file struct pointer (File *f) which was created in pipe_create.
- Check for appropriate error conditions
  - pipe has writers but no data, return EWOULDBLOCK
  - Pipe has no writers and no data, return 0
- Copy the data into the buffer (it s a void *)
  - E.g, You can use memcpy
  - If there is data, Read() returns at most as much data as it was asked for.
  - If there is not enough data, return as much data as the pipe have.
- Delete the data from the pipe s buffer
  - (remove the data that user have just read out or mark the data you have read out as invalid)
- Return number of bytes copied

PIPE_WRITE()
(REFERRED TO AS WRITE() IN PROJECT SPEC)

- Goal: copy data from buffer into the pipe
- Same params as Read(); buffer is the source.
- Implement the buffer like a queue; write appends data, does **NOT** overwrite
- If there is a reader and the pipe has space for data, pipe_Write() returns the number of bytes written.
- Error conditions:
  - No reader, return EPIPE
  - If you choose to implement a fixed size buffer(suggested 32K)): if buffer is full, return 0
  - If you choose to implement dynamically allocated buffer: if malloc() fails, return ENOMEM

PIPE_CLOSE()

REFERRED TO AS CLOSE() IN PROJECT SPEC

- Identify if function is called on the read side or the write side and then act appropriately by closing the side on which it was called.
- Destroy data if there is no reader but there is still data.
- Pipe can also be destroyed if there are no readers and no writers.

VFS LAYER

HOW DOES USER'S READ CALLS PIPE_READ

- The call sequence from user is the following:
  - read()(src/user/pipe-p1.c)→interrupt, context switch→sys_read()(src/geekos/syscall.c)→ Read(src/geekos/vfs.c)→ Pipe_Read()(src/geekos/pipe.c)
  - You may want to read over those function after context switch to help you debug your code.
  - Pay attention to how file ops are used.
- Same routine for write and close.

TESTING

- We provided pipe-p1, pipe-p2, and pipe-p4 programs that you can execute in GeekOS
- Check src/usr/pipe-p1.c for the test details.
- You are encouraged to write your own test.