Lab 3 Design Doc: Pipes

Overview

The goal of this lab is to implement pipes so that two processes can communicate sequentially (one writing and another reading).

Major Parts

File operations: We will modify the file struct so that we can account for the different behavior of reading and writing to a pipe.

Pipe implementation:

- Read
    - Several processes should be able to read from a pipe.
    - A process should block (condvar_wait) until there are bytes to read
    - The processes can do a partial read (fewer bytes than requested).
- Write
    - A process should signal when it starts writing, so read can stop blocking.
    - Write should support multiple processes trying to write to the same pipe (locks).
- Close process communication.

In-depth Analysis and Implementation

File operations

Bookkeeping

We will add a void *info field to the file struct. The info pointer will have additional information only if a pipe is created (initialized after calling fs_alloc_file).

The info pointer will point to a struct pipe pointer with the given fields:

- Synchronization primitives (locks and condition variables).
- A buffer (char array) to hold bytes written to the pipe.
    - The buffer maximum is 512 bytes (kmalloc).
- Status information (such as whether each end is open).

We will also declare and initialize (bbq_init) a global bbq data structure.

Pipe implementation

We will adapt the code from here as a baseline for our blocking bounded queue, which will be used to store char bytes (we will kmalloc 512 bytes for buff). This will be our baseline data structure to implement read and write pipe operations.

Read (pipe_read)

- Read calls bbq_remove(*q) onto the BBQ, which will wait until there's something to read, then remove the buff from bbq.
- If the requested bytes to read are bigger than the stored data, we do a partial read by returning the remaining bytes left in buff.
- Read-only files don't need synchronization (multiple processes can read at the same time).

Write (pipe_write)

- Write will call bbq_insert(*q, buff), which will block until there is room in the internal buffer.
- Return buff data.

Close (pipe_close)

- Call fs_close_file in both fds*.

sys_pipe and Write/Read synchronization

- Validate fds[0] and fds[1], return ERR_FAULT if either address is invalid.
- Return ERR_NOMEM if the two new file descriptors are not available (alloc_fd).
- Proc "wants" to write (open write file descriptor), but there is no open read descriptor (open_files[fds[0]] is NULL).
    - Return ERR_END.
- Proc "wants" to read (open write file descriptor), but there is no open write descriptor (open_files[fds[1]] is NULL).
    - Call pipe_reaad, which will return any remaining buffered data or 0 if no buff data (EOF).
- Call pipe_read with open_files[fds[0]] and pipe_write with open_files[fds[1]]
    - If both run successfully:
        - Call pipe_close.
        - Return ERR_OK.

Risk Analysis

Unanswered Questions

- *In pipe_close, should we call fs_close_file on read or/and write fds?
- When should we call bbq_free?

Staging of Work

First, we will modify the file struct to support file operations. Then we will adapt the code from the bbq queue to store char bytes instead of ints. Finally, we will implement read and write.

Time Estimation

- File operations (10 - 30 minutes)
- Pipe implementation (4-5 hours)
- Adapt bbq (1-2 hours)
- Edge cases and error handling (5 - 10 hours)