Lab 2 Part 2 Design Doc:

Names: Aadhithya Arun, Jacob Young

Overview

Create the pipe object to allow for inter-process communication, exec() to run programs, we need to add synchronization to open files beforehand so that creating pipes doesnt overwrite information in the open file table.

Part 2: Pipe & Exec

Major Parts

Pipe

- Goal: cleanly deploy pipe objects allowing read and write operations
- Challenges: concurrency with pipes, protecting data in the buffer, communication between readers and writers

Exec

- Goal: allow programs to be ran from a function call in a process
- Challenges: argument validations, setting up the stack

Interactions

- Pipe will rely on file_open, file_close, file_dup, to deal with the objects
- Exec will use vspacefree, and other vspace functions to set up a new virtual address space for running the program

In-depth Analysis and Implementation

Pipe

Functions To Implement & Modify

This section should describe the behavior of the functions in sufficient detail that another student/TA would be convinced of its correctness. (include any data structures accessed, structs modified and specify locks used and critical sections).

- Define the pipe struct:
  - buffer (char *): pointer to the start of the buffer
  - Read_offset (int) : number of bytes read (location is buffer + read_offset)
  - Write_offset (int) : number of bytes written (location is buffer + write_offset)
  - available_bytes(int): number of bytes in the buffer (i.e. a limit)
  - read_open(bool): whether any reader is still pointing to the pipe
  - write_open(bool): whether any writer is still pointing to the pipe
  - active_writer(void *): channel where the current process writing to the buffer is sleeping (can use &pipe + 1)
  - Reader_chan (void *): the channel where any blocked readers sleep on while waiting for a write (can use &pipe)
  - Pipe lock (spin lock): lock to determine who has access to the pipe metadata
- implement sys_pipe:
  - Check that the arguments are safe with argptr()
  - Call pipe_open
- pipe_open:
  - get 2 open idx in the global file_info array (if no space return -1 or block?), init one to read-only, the other to write only
  - get 2 fds in the procs fd table, if not enough space return -1, set one to point to read (store in the res[0], where res is the int[] passed in), set the other to point to the write, (res[1])
  - Create the pipe struct
    - Use kalloc to allocate the page, (assume we use the entire page)
    - Can init the read/write offsets to 0, # of available bytes to page table - size of pipe struct, read_open and write_open to true, new lock and condition vars, active writer to NULL or -1 (data type for active writer?)
  - Set both file_infos to point to the pipe
  - Return res
- pipe_read:
  - Acquire the lock for the readonly file_info struct Make sure the file_info struct is actually to a pipe object, acquire the pipe lock
  - Read bytes into the buffer until either read_offset == writeoffset, or

we have read n bytes
  - Send a wake up to the active writer
  - Return whether we have read n bytes or not
- pipe_write:
  - Acquire the lock for the write_only file_info struct, Make sure the file_info struct is actually to a pipe object, acquire the pipe lock
  - Write bytes from the parameter buffer until either we reach the end or, write offset == read_offset if read_open is false return -1
  - otherwise sleep on the pid of the active write, until we are woken up again, and repeat the process
  - Return once all bytes have been written or read_open is false, send a wakeup to the shared reader channel where any readers will be sleeping on
- pipe_close:
  - Close like normal, acquire the lock with the corresponding file_info struct, acquire the pipe lock,If a reader/write, and ref_count of read/write file_info struct = 0, set read_open/write_open to False. If setting read_open to false send a wakeup to the active writer

Design Questions

Pipe Creation

`pipe` returns 2 file descriptors: one for read end and one for write end. How many open files should be allocated as a result of `pipe` call? Can the read end and write end point to the same open file?

- There should be two file_info structs created by pipe(), and they must be separate to ensure permissions still apply to the readers and writers

The `pipe` syscall should dynamically allocate a kernel buffer used for pipe data. How do you dynamically allocate kernel memory in xk? What is the size of the allocated kernel memory? What is the size of your pipe metadata & data?

- We use kalloc to dynamically allocate the kernel buffer, which is 1 page of memory or 4096 bytes
- The size of the pipe metadata and data will also be 4096 bytes, as we will use the first part of the page to store the metadata, while the rest will be the actual bounded buffer.

Pipe Read

Once a `pipe` is created, a process can invoke `read` with the read end. Just like reading a normal file, the read offsets advances upon each read. What metadata must a pipe track to support this?

- The pipe should track the read offset relative to the beginning of the buffer. The reader will hold the pipe lock, so updates to the read_offset are safe.

Although we allow for partial reads, a read from an empty pipe should still block. What metadata must a pipe track to support this?

- We can use the write offset to ensure that empty reads are blocked, if the read_offset == write offset, there is no new information to read so the reader will send a wakeup to the active write and block. In the initalization case, read and write offsets will both be 0, so the reader should block.

Pipe Write

Invoking `write` on a pipe write end should advance the write offset upon each write. What metadata must a pipe track to support this?

- The pipe needs to track a write offset relative to the pipe. Any active writer will have access to the pipe lock and can update the write offset safely

When there is no room to write in the buffer, a writer should block until there is space to write. Is it always true that some space in the buffer will become available? Please explain.

- No, if read_open is not true we should return -1. This could occur after the writer blocks so on file_close readers should send wakeups to writers

In addition, `write` on a pipe must be complete (all requested bytes must be written unless there are no more read ends). How does your design support write sizes larger than the size of your pipe buffer?

- Once the write offset = read_offset we should sleep on the active_writer channel, until a read occurs, once the reader finishes it will send a wakeup to the writer, the write will repeat this until it has written all the bytes

Lastly, `write` must be atomic (no interleaving writes). How do you ensure that a new writer cannot write until the current writer is done? What if the current writer is blocked?

- Writers must have the lock for the file_info struct

Pipe Close

How do you plan to track the life time of a pipe? When can a pipe be deallocated?

- We use the refcounts of the read_only and write_only file infor structs to determine read_open and write_open which determine the lifecycle of the pipe.
- Pipe can be deallocated when both !read_open and !write_open

Is it important to distinguish between a close on a read end vs a write end of the pipe? Why or why not?

- On close we should check the permissions and use then to determine whether to set read_open or write_open to false if the refcount is decreased to 0.

What happens if the last read end calls `close` while there are still blocking writers? How does your design avoid this situation?

- The close will send a wakeup to the active writer, The writer will see there is no space and that read_open = false and will return -1, each waiting writer will then acquire the lock for the file_info struct then perform the same check and return -1.

What happens if all write ends are closed while there are still blocking readers? How does your design avoid this situation?

- If write_open = false and write_offset = read_offset, we return an error, each reader will check this and return 0.

Corner Cases

- Read ends all closed while writers blocked
- Write ends all closed while readers blocked

Test Plans

- Lab2 tests
  - Pipe_test: We will check the arguments to make sure no faulty parameters, and then we make sure to not allow stats on pipe objects, we make sure to initialize the fileinfo_structs with the correct permissions, we then utilize our read and write pipe code to execute the rest of the test.
  - Pipe_closed_ends: Before we set readers or writers to sleep we make sure to check if the opposite end variable is closed or not, if closed, we return, preventing any eternal sleeps of writers waiting for readers or readers waiting for readers
  - Pipe_fd_test: we prevent pip_open when 2 fds are not available for the process, we make sure that we can't read when write ends are all closed
  - Pipe_concurrent: We use locks to prevent corrupted data, and utilize montitor methods to ensure all writes are atomic, we also make sure that readers never reread bytes by updating the read offset synchronously using the pipe lock

Exec

Functions To Implement & Modify

This section should describe the behavior of the functions in sufficient detail that another student/TA would be convinced of its correctness. (include any data structures accessed, structs modified and specify locks used and critical sections).

- sys_exec
  - Validates arguments using argptr and argstr Calls exec
- kernel/sysexec.c
  - Ensure memory alignment
  - Set up a new address space with vspaceinit and vspaceloadcode
  - Copies arguments onto the new stack with vspacewritetova
  - Updates registers and installs the new address space

Design Questions

`exec` takes in a string path and a string array `argv`. How do you plan to traverse through the null terminated `argv`? What validation must you perform during the traversal?

- Loop until we reach a null value, we always need to check the next ptr is valid, and that the string is valid too

`exec` sets up a new address space for a process and frees the old address space. When is it safe to free the old address space?

- After we have loaded the code from the old address spaces and ran vspaceinstall

When copying arguments onto the new stack, which address space is in use? Explain why.

- We still are in the original address space until we run vspaceinstall

Corner Cases

- Ensure argv is properly aligned in memory before execution
- Invalid memory addresses in arguments
- Readers or writers closing unexpectedly

Test Plans

- Lab 2 tests
  - Exec_bad_args
    - Validates incorrect arguments handling (null ptrs or invalid memory addresses)
    - Tests that program fails gracefully
  - Exec_ls

    - Execute the ls command and verify the output lists directories/files correctly
- Exec_echo
    - Run the echo command with various args and check that the output is correct
- Exec_memory_leak
    - Fork a child process and run exec and then check the system memory usage before and after execution
    - Memory leak checks
    - Validate memory that is allocated is properly freed

Risk Analysis

Unanswered Questions

- Will calling wakeup to a channel with nothing sleeping on it error in any way?

Staging of Work

- Implement pipe_open, pipe_read, pipe_write, and pipe_close
- Implement sys_pipe
- Implement exec
- Implement sys_exec

Time Estimation

- Pipe (5-6 hours)
- Exec (5-6 hours)
- Edge cases and Error handling (6-8 hours)