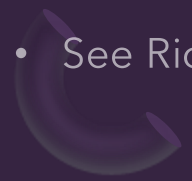
CS 170

Ryan Wenger

File Descriptors – Quick Overview

- The `int fd` parameter to read/write/close/dup/etc is an index into an array stored within each PCB. This can be any size you want, but I made mine 64 entries, which is more than enough.
- This array usually holds pointers to "File" objects that can somehow represent the three devices we're allowed to perform I/O on: pipes, console, and keyboard. If an entry in the array is NULL, then it's unused.
- Because files can be referred to by multiple entries in a PCB's fd table—as well as by multiple PCB's, in the case of fork()—**the objects need to track a reference count** (look it up if you've never heard of "reference counting." It's exactly what it sounds like.)
- Likewise, file objects should be globally allocated (i.e., via malloc). A process could create a file object representing a pipe, and then fork and exit, which should leave the pipe still accessible to its child.
- See Rich's system call lecture notes for more details



Software Layering

- Read, write, and close system calls need to be able to operate on multiple kinds of I/O objects (console and pipes)
- Advise creating read_console()/write_console()/close_console() and read_pipe()/write_pipe()/close_pipe() helper functions that take care of performing the read/write/close operation (close_console can be a no-op). This way, do_read()/do_write()/do_close() can just handle argument unmarshalling & error checking, then simply call out to the appropriate helpers to perform the the operation.
  - Also means that you can unit test your pipe helper functions by calling them from a “regular” Linux C program.

File Objects – Two Implementations

1. Use inheritance/polymorphism: create a base File class with method stubs for read, write, and close, and then subclass it to implement Pipe/Console/Keyboard classes that override and implement the necessary methods.

2. Use a flag variable/enum in each File object to identify its type, then from within sys_write, call a corresponding pipe_write/console_write/etc helper accordingly.

Either way, your system calls need to be able to perform I/O for any (valid) File object, so you should design it to encompass all types of concrete devices that can be referred to by a file descriptor.

Polymorphism in C (for those interested)

- Here is one way (of many) of using polymorphism to implement a File "base/abstract class" and Pipe "derived class" in plain C. The Pipe overrides File's read, write, and close() methods with its own.
  - (note do_read and do_close are incomplete here)

- Likewise, you would then create console_read/console_write functions and attach those to the File objects which refer to standard in, out, and error

- This means that the top-level system calls (read, write, close) don't have to know or care at all what type of device they're operating on.

```c
struct File {
    void *this; /* this/self/me variable in OOP */
    int refcount;
    /* each of these takes a `this` pointer as its 1st argument, then
       whatever else needs to be passed to the corresponding operation */
    int (*read) (void *, char *, size_t);
    int (*write) (void *, const char *, size_t);
    int (*close) (void *);
    /* etc. */
};

struct Pipe { /* etc */ };

/* operate on Pipe pointed to by `this` */
int pipe_read(void *this, char *buf, size_t size) { /* ... */ }
int pipe_write(void *this, const char *buf, size_t size) { /* ... */ }
int pipe_close_reader(void *this) { /* ... */ }
int pipe_close_writer(void *this) { /* ... */ }

struct Pipe my_pipe = { /* ... */ };

/* example file descriptor object for the read-end of my_pipe */
struct File pipe_read_end = {
    .device = &my_pipe,
    .refcount = 1,
    .read = pipe_read,
    .close = pipe_close_reader,
    /* ... */
};

/* example file descriptor object for the write-end of my_pipe */
struct File pipe_write_end = {
    .device = &my_pipe,
    .refcount = 1,
    .write = pipe_write,
    .close = pipe_close_writer,
    /* ... */
};

struct PCB {
    /* ... */
    struct File *fd_table[64];
    /* ... */
}

/* read system call handler (pseudocode) */
void *do_read(int fd, char *buf, size_t size)
{
    /* check errors, buf within user's memory, etc. */
    struct File *target = calling_pcb->fd_table[fd];

    /* ensure this is the read-end */
    if (target == NULL || target->read == NULL)
        syscall_return(-EBADF);

    target->read(target->this, buf, size);
}

/* close pseudocode */
void *do_close(int fd)
{
    struct File *target = calling_pcb->fd_table[fd];
    if (target == NULL)
        syscall_return(-EBADF);

    if ((--target->refcount) == 0)
        target->close(target->this);
}
```

Note on Reference Counting
(this is just one way to do it)

- "File" objects contain reference counts that track how many pointers to that object exist across all PCBs' file descriptor tables
  - Dup/dup2, close, exit, and fork operate directly on ONLY these counts
    - Possible for multiple PCB's to contain pointers to the same File object
  - When refcount reaches 0, call the File's ->close() operation (e.g. pipe_close_writer()) & free() the File object
- Pipe objects contain a count of the number of readers and a count of the number of writers
  - These track the number of distinct "File" objects that refer to the read/write end of the pipe
    - Only ever 0 or 1
  - E.g., pipe_close_writer() decrements the pipe's writers_count. If writers_count is zero, cancel any pending read()s and return EOF to any future read(). If readers_count is *also* zero, free() the pipe object.

pipe(), dup(), dup2()

- Read the man pages
- Very straightforward syscalls to implement (although the Pipe objects themselves are certainly not)
  - Advise creating some helper classes:
    - sema_q – the “semaphore-controlled” queue; identical to the one from the reader thread in lab 1. Generalize this and use it for pipes, console read, and (optionally) console write (more on that later)
    - “File” – type that is pointed to by each entry in a PCB’s file descriptor array. Can represent pipe read-end, write-end, console output, and console input devices. Holds a count for the number of references to itself so it knows when to release the underlying object (pipe, in our case).
    - Anything else you deem necessary (pipes, custom locks, etc.)

Read/write Atomicity of Pipes

From Rich's hints: Imagine that

- writer 1 executes write(pd[1],buf,10)
- writer 2 executes write[pd[1],buf1,10)
- reader 1 executes read(pd[0],rbuf,10)
- reader 2 executes read(pd[0],rbuf1,10)

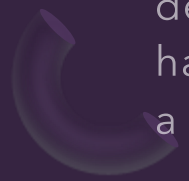
The funny thing about pipes is that regardless of the order in which these reads and writes happen, the pipe will attempt to ensure that one of the two following conditions are true after all four system calls have completed (regardless of their execution order). Either

- rbuf contains the contents of buf and rbuf1 contains the contents of buf1, or
- rbuf1 contains the contents of buf and rbuf contains the contents of buf1

➔ Thus, use mutexes to limit access to each end of a pipe to one process at a time, just like what was done for the console in lab 1

Kicking blocked readers/writers from pipes

- Pipes with readers and no writers should indicate EOF to any read() operations
  - Likewise, write() calls on pipes with writers but no readers should return EPIPE
- What if a read() is in progress, waiting for more bytes, when the final writer closes the write end? We need to “kick” the reader so it doesn’t block indefinitely and immediately terminates the operation.
  - How you accomplish this depends on how your pipe/queue abstractions work, so keep it in mind when designing them. Probably you’ll need to V() a semaphore one or more times to wake each blocked reader, then have it check whether any writers still exist before continuing to retrieve bytes. (Same goes for writers blocked on a full queue when the final reader closes its end, except you’ll also need to return EPIPE to the user).



Possible Plan of Attack

1. Add file descriptor table to PCB structure (can be an array of pointers to File objects)
    - Generalize existing read, write, and close system calls to operate on any entry in this file desc. table. Don't worry about implementing pipes yet, but make sure you have a mechanism to call appropriate logic for pipes vs. console I/O.
    - After refactoring, make sure functionality from lab 2 still behaves the same as before.
2. Implement dup & dup2, and modify close, exit, and fork to operate on File reference counts
    - None of these should malloc() anything file-related. Dup() and fork() should simply copy pointers and bump reference counts; close and exit should decrement reference counts and (possibly) free() the File object
3. Implement pipes. You want a struct Pipe, as well as helper functions to read, write, and close either end of a Pipe.
    - Testing the logic of pipes in a standalone program/unit testing framework is a good idea here, since you can trigger edge-case behavior much more directly than when running inside the simulator.
4. Hook up pipes to the entries in your file desc. table
    - Enable File objects to hold pointers to a Pipe object and call the pipe I/O functions from do_read and do_write
5. Implement pipe() system call. Here is where you should malloc 1 Pipe object and 2 File objects (1 for the read end and 1 for the write end).
6. Test.

Some gotcha's to look out for

- Stdin/stdout/stderr can be dup'ed, so you can't always assume that any fd greater than 2 in a read/write/close call refers to a pipe
- Reading from a pipe should *only* return EOF if there are no more writers. If the pipe is empty, then the read should block and wait for data.
- To prevent interleaving of successive write(x, x, 1) calls (e.g. due to printf), implement a queue for the console, along with a console worker thread to push the queue's contents out to the console device.
  - sys_write calls operating on stdout should instead push bytes into this queue, allowing back-to-back write calls to be nonblocking (up to the size of the queue's buffer), and thus not be interleaved with another process's
  - Be careful that you don't SYSHalt() while this queue isn't yet empty—you may have to add an extra check into your shutdown logic to prevent this from happening

Other Tips

- Don’t forget to decrement reference counts of every (valid) entry in a PCB’s file descriptor table when it exits and increment them when it forks
- It’s a very good idea to implement your pipes in a way that allows you to directly unit-test them without needing to go through KOS/the simulator
- Use the console read queue implementation from lab 1 for pipes. Each pipe gets its own queue with a buffer, head/tail index, nslots/nused semaphores, and mutex/mutual exclusion lock