CS 170

Ryan Wenger

File Descriptors – Quick Overview

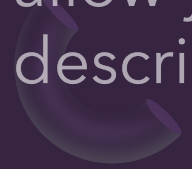
- The `int fd` parameter to read/write/close/dup/etc is an index into an array stored within each PCB. This can be any size you want, but I made mine 64 entries, which is more than enough.
- This array usually holds pointers to "file" objects that can somehow represent the three devices we're allowed 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 (whether through malloc or file-scope declaration in the .data segment). 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.

File Objects – Two Implementations

1. Use polymorphism: create a parent 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.

Choice is up to you (though #1 is cleaner). Either way, the File object should allow you to encompass all types of underlying devices that can be stored as file descriptors.



Example File Object Definition

- Don’t use this code verbatim—it’s untested and I wrote it very quickly just as an example. But it illustrates one way of using polymorphism to implement the File class in valid C. Note that there are several ways to design polymorphism/inheritance in C, and each has its own tradeoffs.

```c
typedef void * Device;

struct File {
    Device this; /* this/self/me variable in OOP */
    int refcount;
    int (*read) (Device this, char *, size_t);
    int (*write) (Device this, const char *, size_t);
    int (*close) (Device this);
    /* etc. */
};

struct Pipe { /* etc */ };

int pipe_read(struct Pipe *this, char *buf, size_t size) { /* ... */ }
int pipe_write(struct Pipe *this, const char *buf, size_t size) { /* ... */ }
int pipe_rclose(struct Pipe *this) { /* ... */ }
int pipe_wclose(struct Pipe *this) { /* ... */ }

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

/* file descriptor object for the read-end of my_pipe */
struct File example_pipe1 = {
    .device = &my_pipe,
    .refcount = 1,
    .read = &pipe_read,
    .close = &pipe_rclose,
    /* ... */
};
/* file descriptor object for the write-end of my_pipe */
struct File example_pipe2 = {
    .device = &my_pipe,
    .refcount = 1,
    .write = &pipe_write,
    .close = &pipe_wclose,
    /* ... */
}

struct PCB my_pcb = {
    /* ... */
    .fd_table[3] = &example_pipe1,
    /* ... */
}

/* read system call handler */
sys_read(int fd, char *buf, size_t size)
{
    /* ... */
    struct File *target = my_pcb.fd_table[fd];
    target->read(target->this, buf, size);
    /* ... */
}
```

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 console write*
    - fd_object/File – type that is stored in each PCB’s file descriptor array (usually as pointers). Can represent pipe read-end, write-end, console out, and console in devices. Holds a count for the number of references to itself so it knows when to release the underlying object (pipe, disk file, etc.).
    - 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

Some gotcha's to look out for

- One end of a pipe may be closed while an operation on the other is in progress. For example, if a write operation is blocked due to a full pipe buffer and all of the read ends close, the writer should abort (and vice versa).
- 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, though, 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