

# Project 0: Pipe

Consult the submit server for deadline date and time

You will implement anonymous<sup>1</sup> pipes by implementing a version of the Pipe() system call and related calls Read(), Write(), and Close().

The primary goal of this assignment is to ensure that you have a working development environment for programming GeekOS and to familiarize you with some important files.

## 1 Desired Semantics

In general, Pipe() here should work as described in the man page for pipe (pipe(2)), with the following exceptions and specifics.

- Pipe() takes two arguments: each a pointer to an integer location. When Pipe() returns successfully, it would have created a pipe and filled the two locations with file descriptors (integers), one pointing to the reading end of the pipe and the other to the writing end of the pipe. This is in contrast to pipe(2), which fills in an array of two elements.
- Read() takes three arguments: a read file descriptor, a pointer to the buffer into which data is read, and the number of bytes to read. Calling Read() on a pipe returns immediately, i.e., it is non-blocking. If there are writers, but no data, Read() returns EWOULDBLOCK. If there is data, Read() returns at most as much data as it was asked for in the third parameter: it does not wait for as many bytes as were asked for if they are not available. If there are no writers (i.e., the writing file descriptor has been closed) and there is no data, Read() returns 0.
- Write() takes three arguments: a write file descriptor, a pointer to the buffer holding the data to write, and the number of bytes to write. Calling Write() on a pipe returns immediately, i.e., it is non-blocking. If there is no reader, Write() returns EPIPE. (In reality, it also delivers SIGPIPE to the writer, but we don't have signals yet.) If there is a reader and the pipe has space for data, Write() returns the number of bytes written. (Depending on the available space, this may be less than the number of bytes in the Write() request.) If no bytes can be written because the pipe's buffer is dynamically allocated and Malloc() fails, Write() returns ENOMEM. If no bytes can be written because the pipe's buffer is a fixed-size buffer (suggested size 32K) and there is no space, Write() returns 0.
- Close() takes one argument: a (read or write) file descriptor. Calling Close() on a pipe closes the end of the pipe indicated by the file descriptor. If the pipe has data and there is still an active reader, the pipe (and its data) is not destroyed. If there is data and no reader, the data can be destroyed.
- A pipe conveys a stream of bytes, without regard to the chunks in which the bytes were written. For example, if a writer writes 2 bytes then 3 bytes, and a reader requests to read at least 5 bytes, the reader should get 5 bytes all at once.

## 2 Future Changes

The next assignment will be to support Fork(), which works in concert with Pipe to permit two processes to communicate with each other. To do so, Fork() must increment the reference counts associated with open file handles so that when they are closed in one process, they remain open in the other.

---

<sup>1</sup>Anonymous pipes are the typical kind. The alternative are “named” pipes, which are not often used.

## 3 Background

Shankar and Jeff wrote a detailed overview of GeekOS source at <https://www.cs.umd.edu/~hollings/cs412/s17/GeekOSoverview.pdf>.

### 3.1 Getting Started with GeekOS

See [https://www.cs.umd.edu/class/fall2018/cmsc412/setup\\_proj\\_env.html](https://www.cs.umd.edu/class/fall2018/cmsc412/setup_proj_env.html)

### 3.2 General

In the build/ directory, you will find various Makefiles. If you want to alter how GeekOS is built in its entirety (e.g., adding source files), edit Makefile.common. If you want to alter how GeekOS is built on your machine, edit Makefile.linux or Makefile.darwin or Makefile.linux.x86\_64 as appropriate. Makefile.common is consulted on the submit server. The rest are not.

NOTE: You shouldn't ever need to add new source files.

In the src/ directory, geekos/ contains kernel code, user/ contains user programs, common/ contains code linked to both, and libc/ contains library code for user programs. The tools/ directory has programs run during compilation. You will spend most of your time in the src/geekos/ directory.

The include/ directory is similar: geekos/ for kernel specifics, libc/ for user-included code. Note that there is no Malloc() in user code; there is no Brk() system call to extend the heap.

### 3.3 Debugging GeekOS

In the build/ directory, have two terminal windows available to you. You may use GNU screen or tabs in your terminal to achieve this. In one terminal, run 'make dbgrun': This runs qemu so that it waits for the debugger to attach. In another, run 'make dbg'. In the debugger console, type "c" to continue execution. When you want to set a breakpoint, hit Ctrl-C and type "break Sys\_Pipe", for example, then "c".

Key, in practice, is to use KASSERT() extensively and Print() judiciously. Assert any assumption for which violation would be egregious: for example, if a reference count exceeds 1000, implying it was uninitialized, or is negative, implying it was decremented without check.

Assert any assumption that would cause your code to crash, for example, that a pointer passed as a parameter is not NULL. The address 0 is perfectly valid in hardware: dereferencing NULL will not crash, it will simply fail to work as well as you'd like.

### 3.4 System Calls in GeekOS (and in general)

To invoke a system call, the caller places the system call number in register eax, places parameters in the other registers, and invokes an interrupt. That interrupt wakes up the kernel, causing the user's registers to be stored on the kernel's stack while the kernel executes. The kernel (trap.c) finds the system call function (pointer) in a table indexed by eax and invokes that function.

(In the current version of GeekOS, all interrupt handlers (including traps, exceptions, system calls) run with interrupts disabled. System calls are invoked with interrupts enabled.)

When the system call's function returns, trap.c places the return value in the memory location where eax was stored on the kernel stack. Eventually, the system call will "return" by popping these registers off the stack. ("Eventually" because other processes may run in the meantime.)

### 3.5 Pipes, the vfs layer, and more.

Each pipe has a writer and reader side, which, in general but not in this project, may be written to by one or more writers and read from by one or more readers.

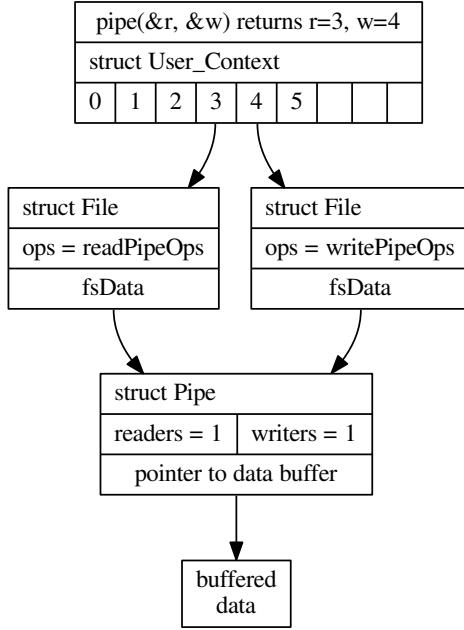


Figure 1: Illustration of the file descriptor table (top), where entries point to struct File objects. There are two struct File objects for the pipe. The pipe object must track whether there are any readers or writers and keep buffered data.

Recall from 216 that a file *descriptor* is an integer index into an array of pointers to open *file descriptions*. See the man page for open(2) for more information.<sup>2</sup>. That file description in turn references an inode (or other object that represents the actual file) for the file that is opened. More than one process can open the same file independently and will have different positions within that file: the file position is stored in the *file description*, and independently opening the same file will result in more than one file description pointing to the same inode. If a process forks a child, both share the same file descriptions at the same, original, descriptor index. The file descriptions have an incremented reference count, but both parent and child share the current position in the file.

In geekos, the file description is a 'struct File'. Each File has its own methods for reading, writing, closing, seeking, etc. This permits files of different types to be made: e.g., /proc file system entries, /dev device files, or even files on different file systems. These are the "ops" in the File structure. You might compare this to interface inheritance, in which a "File" can be implemented by a GOSFS file, a FAT file, a GFS2 file, or even a Pipe. The methods that implement this interface are specified dynamically.

The operations on a Pipe() are limited: One cannot Seek() in a pipe. One cannot Read() the writing side of a pipe, or Write() the reading side of a pipe.

The 'struct File' also has an 'fsData' element for storing file system specific data associated with this file. Use this pointer to store the Pipe and its data.

---

<sup>2</sup>"A call to open() creates a new open file description, an entry in the system-wide table of open files. This entry records the file offset and the file status flags (modifiable via the fcntl(2) F\_SETFL operation). A file descriptor is a reference to one of these entries; this reference is unaffected if pathname is subsequently removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via fork(2)."

### 3.6 Copy\_To\_User and Copy\_From\_User

User applications have pointers to addresses in their address space: on their stack or in their global variable space. When a user application passes an address, say 0x1000, to the kernel as an argument, the kernel has to fetch the stuff at the *user's* 0x1000, not the *kernel's* 0x1000.

There are two functions to help with this, `Copy_To_User()` and `Copy_From_User()`. A third generally function, `Copy_User_String()`, invokes `Copy_From_User()` to copy a string. These functions translate the user-visible address (which is a virtual address because of segmentation) into a physical (for now) address that the kernel can use, then copy from that physical address to another physical address in the kernel. If the user-provided address is invalid (beyond its address range), `Copy_From_User` fails rather than grab data from some arbitrary place in memory.

Eventually, you will modify these functions (slightly) to work with paged virtual memory, so having this functionality in only one place is a good thing.

### 3.7 Rules and Hints

Use, but do not alter 'struct File'. For the next project, struct File will need a reference count, but not now, since there is no Fork.

You may modify any function needed, regardless of whether a `TODO_P(PROJECT_PIPE)` macro is present. (You may need to modify other functions.)

There is no `Realloc()`. We are unlikely to test behavior when memory is exhausted.

You may limit the pipe buffer size to 32K.<sup>3</sup>

Most, if not all changes will be in `pipe.c`. You may want to trace through `Sys_Write`, `Sys_Read`, and `Sys_Close`; these will transfer control to the functions described in `vfs.c`, which in turn will invoke the ops.

Do not allocate large buffers on the GeekOS stack, it is of finite size and there is no protection against overflow. A 256 character buffer such as for a filename is probably fine; larger should be allocated elsewhere.

## 4 Tests

There are two public tests: `pipe-p1` and `pipe-p2`. They are intended to cover the bulk of the functionality described here.

Expect "secret" tests to exercise other behavior described in this handout. "Secret" tests are likely to cover bizarre mistakes such as having only one global pipe and bad memory handling via repeated invocation of Pipe and Close.

---

<sup>3</sup>32K appears possible, but I have chosen it arbitrarily.