Project 0: Pipe

Due 11:59 PM, September 18, 2013

In this project, you will create an anonymous pipe[1]. You will implement the Pipe() system call, and the 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 introduce you to some of the source code.

Instructions for getting started with QEMU and GeekOS are at http://www.cs.umd.edu/~shankar/412-Notes/geekos-install-09-05.pdf and http://www.cs.umd.edu/~jonfd/f13-412/09-09-gdb-p0.pdf.

1 System Calls, I/O, and Pipes

1.1 System Calls

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 (using code in src/geekos/trap.c) finds the handler for the system call (as a function pointer) in a table indexed by eax and invokes that function.

It does this with interrupts disabled. Disabling interrupts prevents any other process from taking control of the processor. Any system call that may take substantial time should enable interrupts while not manipulating shared kernel state. It can do so by calling Enable_Interrupts(). Then eventually when it leaves the system call function, it must call Disable_Interrupts(), which restores the interrupt state as it was prior to the Enable_Interrupts() call. While you will not need to enable interrupts in this project, you will in future projects.

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, so that the caller finds the return value in eax. ("Eventually" because other processes may run in the meantime.)

1.2 I/O and Pipes

I/O system calls refer to files with *file descriptors*. Within the kernel, a file descriptor is an integer index into an array of *file descriptions*. (See the man page for open(2) for more information.) That file description in turn points to an object that the OS can use to work with the actual file. For a file on disk, for instance, the description might point to a struct that tells what disk blocks belong to the file.

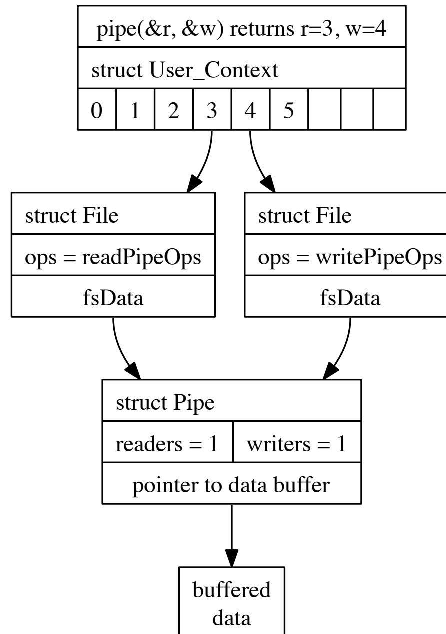
More than one process can open the same file independently. The processes will then have different positions within that file: the file position is stored in the file description, and independently opening the same file will result in more than one file description pointing to the same inode. If a process forks a child, however, 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 the same set of system calls to act on files of different types: not only files on different filesystems such as PFAT and GOSFS (which you will implement later), but entries for processes in /proc, device files in /dev, and, of course, pipes. You might compare this to interface inheritance, in which a List can be implemented by an ArrayList, LinkedList, or another class.

[1] Anonymous pipes are the typical kind. The alternative is a "named" pipe, which is used much less often.

Since C is not object-oriented, we create this machinery by hand. The handlers for I/O system calls such as Sys_Read(), Sys_Write(), and Sys_Close() call functions Read(), Write(), and Close() in vfs.c. The handler passes these functions a pointer to a struct File. The File, in turn, includes a struct File_Ops, which is just a list of pointers to the functions that will actually do the work. The struct File also has an fsData element for storing file system specific data associated with this file. The vfs functions look up the correct operation and call the function from the File_Ops, passing it the fsData. You will use fsData to refer to your Pipe and its data.

Figure 1: File descriptors, descriptions, and the underlying pipe.



Each pipe has a reading and a writing side. The Pipe() system call takes pointers to two integers, and sets those integers to the file descriptors for the reading and writing side of the file (which means that you must set up two file descriptions). The operations on a Pipe are limited: one cannot Seek() in a pipe, Read() the writing side of a pipe, or Write() its reading side. The file `src/geekos/pipe.c` includes preset File_Ops structs that you can use in your file descriptions.

Figure 1 illustrates how the data works together.

2 Requirements and Hints

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

- The call to Pipe() takes two arguments: pointers to integers (file descriptors) for the reading end of the pipe and the writing end of the pipe. This is in contrast to Unix pipe, which fills in an array of two elements.
- Calling Read() on a pipe will return immediately. That is, Read() is non-blocking. If there is data in the pipe (whether or not the writer has closed its end), Read() will return the minimum of (1) the amount of data in the pipe, or (2) the number of bytes requested in the third argument to Read(). If there is no data, read behavior depends on whether the writer has closed its end of the pipe. If the

writer has closed its end, Read() returns 0, indicating the end of data. If the writer has not closed its end, Read() returns the error code EWOULDBLOCK.

- Calling Write() on a pipe will return immediately. If the reader has closed its end of the pipe, Write() will return EPIPE[2]. If there is no memory (a Malloc() fails), Write() should return ENOMEM. Otherwise, Write() should return the number of bytes written. As noted below, you don't have to store more than 32K of written data at a time. Do, however, accurately return the number of bytes you stored in the pipe buffer.
- Calling Close() on a pipe closes only one end of the pipe. If the writer closes its end of the pipe before the reader closes its end, and not all the data has been read, the data must remain available to the reader. If the reader closes its end first, you may destroy the data.
- Reads and writes treat the pipe like a stream of data; they are not individual messages. That is, if we write 2 bytes, then write 3 bytes, then try to read 5 bytes, we will get all 5 bytes at once.

Send your code to http://submit.cs.umd.edu, either by zipping it and uploading it on the Web, or by doing a make submit in the build directory. In the latter case you will need the .submit file that you can download from the submit server.

2.1 Debugging GeekOS

The discussion section slides, linked above, give a description of using gdb to debug GeekOS. In addition, you will want to use the KASSERT macro extensively, and the Print function judiciously.

KASSERT will test that a condition is true, and if it is not, stop execution with a message in the QEMU window. 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.

Also 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.

You can use Print() to print messages to the QEMU window, as well. It works like printf() in C.

2.2 User and kernel space buffers

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 function, Copy_User_String(), is available in syscall.c. It 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 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.

2.3 Miscellaneous notes

You may limit the pipe buffer to 32K. That is, our tests will not write more than 32K before reading. Of course, we may (for instance) write 32K, read 16K, write 8K, read 16K, write 24K, and so forth.

Use the function Malloc() (note the capital M) to allocate memory, and use Free() to release it. There are no analogues to realloc() or calloc() in C. Malloc() and Free() are available only in kernel mode. User code in GeekOS has no dynamic memory allocation.

You will notice the macro TODO_P(PROJECT_PIPE) in several of the functions you need to modify. This macro prints a highlighted message on the screen and halts execution. Once you start working on those

[2] A real operating system would deliver a SIGPIPE, but we don't have signals yet.

functions, you'll want to delete this code. You don't need to worry about TODO_P with other project names; see include/geekos/project.h.

Most of your changes will be in syscall.c and pipe.c. In addition to Sys_Pipe, you'll need to write small versions of Sys_Read, Sys_Write, and Sys_Close. These functions will transfer control to the existing read, write, and close functions in vfs.c, which in turn will invoke the pipe read, write, and close operations. We will provide pseudocode for Sys_Read to get you started.

3 Tests

There are two public tests in the user directory of GeekOS, pipe-p1.c and pipe-p2.c. You can run them by starting GeekOS and typing pipe-p1 or pipe-p2 at the shell prompt. They are intended to exercise most of the functionality described here. Other tests will cover odd mistakes such as having only one buffer shared among all pipes, or failure to free memory once a pipe is closed.

You can write your own tests by creating new files in src/user. The Makefile will automatically compile them and include them in the GeekOS disk image. As mentioned above, you do not have access to malloc() in user mode; to create a buffer, you will need to declare something like `char buf[32768];` as an automatic (local) variable.