Project 1: Fork and Exec

You will implement the Fork() and Exec() operations, preserving the semantics of pipe() across both: that both parent and child have the same file descriptors attached to the pipe, both parent and child may read and write to the pipe, and the exec'd program inherits the file descriptors of the process.

The primary goal of this assignment is to develop an understanding of the process control block (struct kthread) and user context structures (struct User_Context).

1 Desired Semantics

In general, Fork() here should work as described in the man page for fork (fork(2)), with the following exceptions and specifics. GeekOS lacks many of the features that "real" fork() must make decisions about, such as signal inheritance, what to do about multiple threads, and resource limits.

- Fork() must return the child's pid in the parent and zero in the child. The Child's pid must be new.
- Global variables must start out the same, but each process updates its own copy.
- The stack should appear the same at the point of the Fork call, but each process has its own copy.
- File descriptions are shared: when either process updates the position in a file, that should update the position as seen by the other process.
- Fork() should return ENOMEM if a memory allocation fails.

1.0.1 General

```
git clone git://scriptroute.cs.umd.edu/geekos-project
cd geekos-project && git pull
```

NOTE: We will not be using subversion (svn). Any URLs containing "svn" on any documentation you find are obsolete. Instead, git allows you to track and commit changes *locally* without requiring you to run a different revision control system. If you are a master of git, you should be able to create branches for different projects.

Using "git", occasionally you will want to invoke "git pull" to grab any changes we may have made, e.g., to a .submit file or the projects.h file.

1.1 Process Creation.

The existing scheme for creating new processes is the Spawn() function. Spawn() is a bit like the Windows CreateProcess function, which creates a new process with a program name to implement. It does *everything*, which means it contains the core of both Fork() and Exec(). Spawn lacks a facility for creating the intermediate state: copying the address space from parent to child. It also lacks the functionality we seek with inheriting file descriptors. Finally, Spawn() creates the new process with an empty context rather than at the point of a fork. We'll discuss these features in a bit more detail than those you'll find already in the code.

Each process is represented by a "struct kthread." There are kernel processes, which are really threads inside the kernel, since they all share the same address space. User processes have a "userContext" in the kthread: the userContext contains the stuff unique to user processes: their own address spaces, file descriptors, etc.

To Fork(), a new kthread must be created. Unlike in Spawn(), the contents of memory should not be the result of loading a program file, it should be a copy of the memory of the parent. In a real operating system with paging, this copy would occur lazily (and efficiently) via copy-on-write.

1.2 Copying the address space

Look in userseg.c to note how a user context can be created. The kernel has memcpy(). This one is pretty easy.

1.3 Inheriting file descriptors

Copy the file descriptor array from parent to child, then iterate over this list incrementing the reference count of each file. Most file descriptors will be NULL.

1.4 Register Context

When a function makes a system call, the user process's registers are pushed onto the kernel's per-process stack. The kernel's stack is in the kthread "stackPage", while the stack pointer is esp. When cloning a process, the child should have substantially the same kernel stack, to represent substantially the same registers (one will be different).

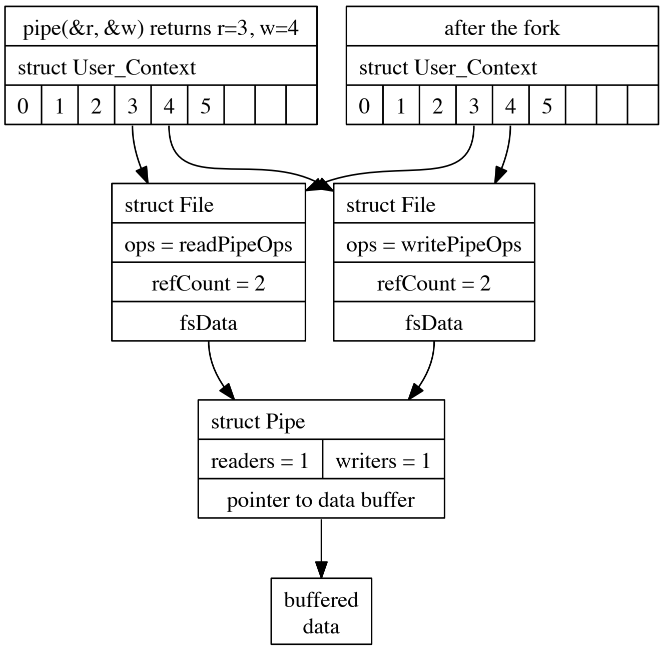

Figure 1: Illustration of the file descriptor table (top), where entries point to struct File objects, after Fork(). There are two struct File objects for the pipe. The pipe object must track whether there are any readers or writers and keep buffered data, the File object must track how many processes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

1.5 Rules and Hints

Follow the path of Spawn() and determine which features are part of Exec() and which features are part of Fork(). Most everything in you need is already present.

Alter 'struct File' to add a reference count. Alter Open(), Fork(), and Close() to set, increment, and decrement this reference count.

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

Most changes will be in pipe.c and syscall.c. Note that you will have to write simple versions of 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.

2 Tests

There are two public tests: fork-p1 and forkpipe. 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 failing to replicate the data segment and failing to count references for files.