Project 1: Fork and Exec

Due Feb 20

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 Kernel_Thread) 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 and sequential.
- 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. We don't have files with positions yet, but for now, pretend that we do.
- Fork() should return ENOMEM if a memory allocation fails.

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 Files and inheriting file descriptors

Copy the file descriptor array from parent to child. Since files can now be shared between processes, you will need to add a reference count in *struct File*. Initialize the reference count in the appropriate places and modify it accordingly for Fork() and Exec().

1.4 Register Context

When a function makes a system call, the user process's registers are pushed onto the top of the kernel's per-process stack. The kernel's stack is in kthread's *stackPage*, while the stack pointer is *esp*. At the very beginning of a system call, *esp* coincides with the *struct Interrupt_State *state* argument, pointing to the location on the stack where the user's registers are stored. However, if, during the system call another interrupt occurs, *esp* will be updated to the current value of the actual esp register. Therefore, the user's registers should always be accessed using the *state* variable, not *esp*.

When cloning a process, the child should have substantially the same kernel stack, to represent substantially the same registers (one will be different).

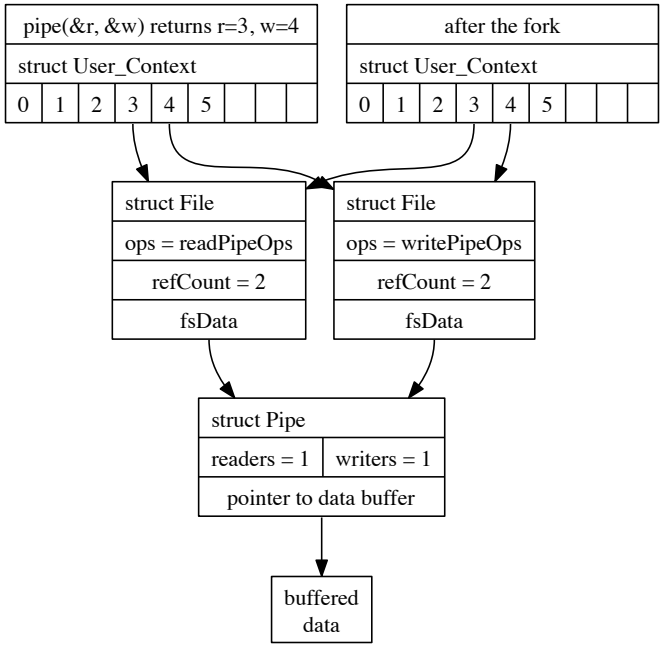

Figure 1: Illustration of the file descriptor table (top), where entries point to struct File objects, after Fork(). There are two struct File objects for the pipe. The pipe object must track whether there are any readers or writers and keep buffered data, the File object must track how many processes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

1.5 Fork Bomb

One of the tests will be a fork bomb test which will rapidly deplete the available memory as more and more processes are created. At some point a Malloc call will fail, and you must handle this appropriately. In addition, you will need to upgrade the *Exit* function in kthread.c.

The *Exit* function is called when a thread terminates. Most threads (non-detached) are created with a reference count of 2, one for themselves and one for their parent. The memory contents associated with a given thread are only released when the reference count is decremented to 0. As currently written *Exit* only releases the reference a thread has to itself. Because the parent reference is never released, all of the memory allocated to the process won't be freed, despite its no longer being needed. You must modify the *Exit* function so that upon termination, a process "informs" all of its children of its imminent demise. This can be accomplished using the *owner* field in the struct kthread, as well as the *s_allThreadList*.

1.6 Synchronization

With Fork(), particularly with SMP, comes the possibility that two of your processes may be running at the same time. If not careful about preventing concurrent access to the same operations, you may create a race condition that will cause data corruption at random; alternately, you may cause a deadlock condition - no processes can make progress.

In this section, I use "process" and "kernel thread" interchangeably, since the code you're writing is in the kernel, running in a thread (they share an address space in the kernel), on behalf of a process.

1.6.1 The hammer

Disable_Interrupts() and Enable_Interrupts() grab and release the "big" global lock. Begin_Int_Atomic() and End_Int_Atomic() grab and release the global lock if it is not already held, allowing routines to be called with or without the lock.

These routines will prevent interrupts from occurring on the current processor and grab a lock that prevents the interrupt handler from running on any other processor. That means that the timer interrupt will eventually block execution of the other processor's activity.

Any process waiting for the global lock spins, with interrupts disabled, expecting the code on the other processor to release the global lock. (That is, it would be bad if the other processor is somehow not going to release the global lock.)

System calls in geekos start with interrupts disabled, but as you can imagine, if that operation requires a device to do something, interrupts have to be enabled.

1.6.2 The knife

Mutex_Lock and Mutex_Unlock (see synch.c) will put a blocked thread on a wait queue until the lock is released. See, for example, the use of mutexes in vfs.c to guard the list of file systems.

Using a mutex requires that interrupts be *enabled*. That is, the calling code expecting to request a mutex has to know that it might not get it immediately. Another thread might need to run, and data structures that might be shared should be in a consistent state.

1.6.3 The rock

The spinlock is quite possibly not a good synchronization tool, but it appears in geekos and may be useful for guarding data structures for brief periods efficiently.

Each list in geekos (see list.h) is guarded by a spinlock. This spinlock is meant to defend the list against accidental concurrent access, but because of its limitations, may not be appropriate protection.

Simple list operations automatically grab and release that lock. Some operations, such as "get next in list" expect the lock to be held. Consult the list.h source to determine which functions are appropriate for you.

From `http://en.wikipedia.org/wiki/Spinlock`,

> Because they avoid overhead from operating system process rescheduling or context switching, spinlocks are efficient if threads are only likely to be blocked for a short period. For this reason, spinlocks are often used inside operating system kernels. However, spinlocks become wasteful if held for longer durations, as they may prevent other threads from running and require rescheduling. The longer a lock is held by a thread, the greater the risk is that the thread will be interrupted

by the OS scheduler while holding the lock. If this happens, other threads will be left "spinning" (repeatedly trying to acquire the lock), while the thread holding the lock is not making progress towards releasing it. The result is an indefinite postponement until the thread holding the lock can finish and release it. This is especially true on a single-processor system, where each waiting thread of the same priority is likely to waste its quantum (allocated time where a thread can run) spinning until the thread that holds the lock is finally finished.

1.7 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 that you need is already present. If possible, reuse of existing functions is preferred to copying over large chunks of code.

Proper implementation of the file reference count will require modifications to Open(), Pipe(), Close() in addition to possible changes in Fork() and Exec().

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

2 Tests

There are a few public tests: fork-p1, forkpipe, and forkexec. 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 pipes.