Jacob Adkins

CSE 5343

Kernel Space vs User Space File I/O Speed

Introduction

The Linux operating system divides itself into two major spaces of operation: the kernel space and the user space. The kernel lives in the kernel space along with other low-level, trusted software such as device drivers. This space allows for highly privileged code execution without the extra overhead of safety checks. The user space, on the other hand, is where most software lives and is the environment from which the user can issue commands.

For software in the user space to access kernel functionality, such as file I/O, it must make a request to the kernel that the action be performed. This request requires several factors to be checked, such as the user's permission level and the file's mode bits, and then it must make its way layer-by-layer through the OS's protection ring until it reaches the privilege level necessary to actually perform file I/O and file system object manipulation. The same file I/O operation, when issued from within the kernel space, should be much faster because it does not need to delegate through as many layers of protection. But exactly how much faster?

Method

A similar file I/O operation was implemented in both the *init_module()* function of a linux kernel module and in a simple c program. The operation opens a file, writes every number from 0 to 100000, and closes the file. Both the module and the program record their own execution time and print it into a log. The *init_module()* function is run every time a kernel module is inserted into the linux kernel, so by inserting and removing the kernel module and running the c program multiple times a sizeable dataset was acquired. Both the module and program were compiled and the data was collected on an Ubuntu 14.04 virtual machine running on a 2015 Macbook Pro 13” laptop. Running the tests in a virtualbox provided a sterile and safe environment, and should not affect the comparative performance between the module and the program.

Below are the important sections of the file I/O operation in the kernel module and the c program. The *file_open(), file_sync(), and file_close()* functions create a file object pointer, sync any changes to a file, and delete a file object pointer, respectively.

Kernel Module:

```
getnstimeofday(&begin);
                oldfs = get_fs();
                set_fs(get_ds());
                fileptr = file_open("/home/file", O_WRONLY|O_CREAT, 0644);

                for (i = 0; i < 100000; i++)
                {
                        sprintf(str, "%d\n", i);
                        vfs_write(fileptr, str, strlen(str), &offset);
                        offset += strlen(str);
                }

                set_fs(oldfs);
                file_sync(fileptr);
                file_close(fileptr);
getnstimeofday(&end);
```

C Program:

```
begin = clock();

                FILE* fp = fopen("file", "w");
                for (i = 0; i < 100000; i++)
                {
                        sprintf(str, "%d\n", i);
                        fputs(str, fp);
                }
                fclose(fp);

end = clock();
```

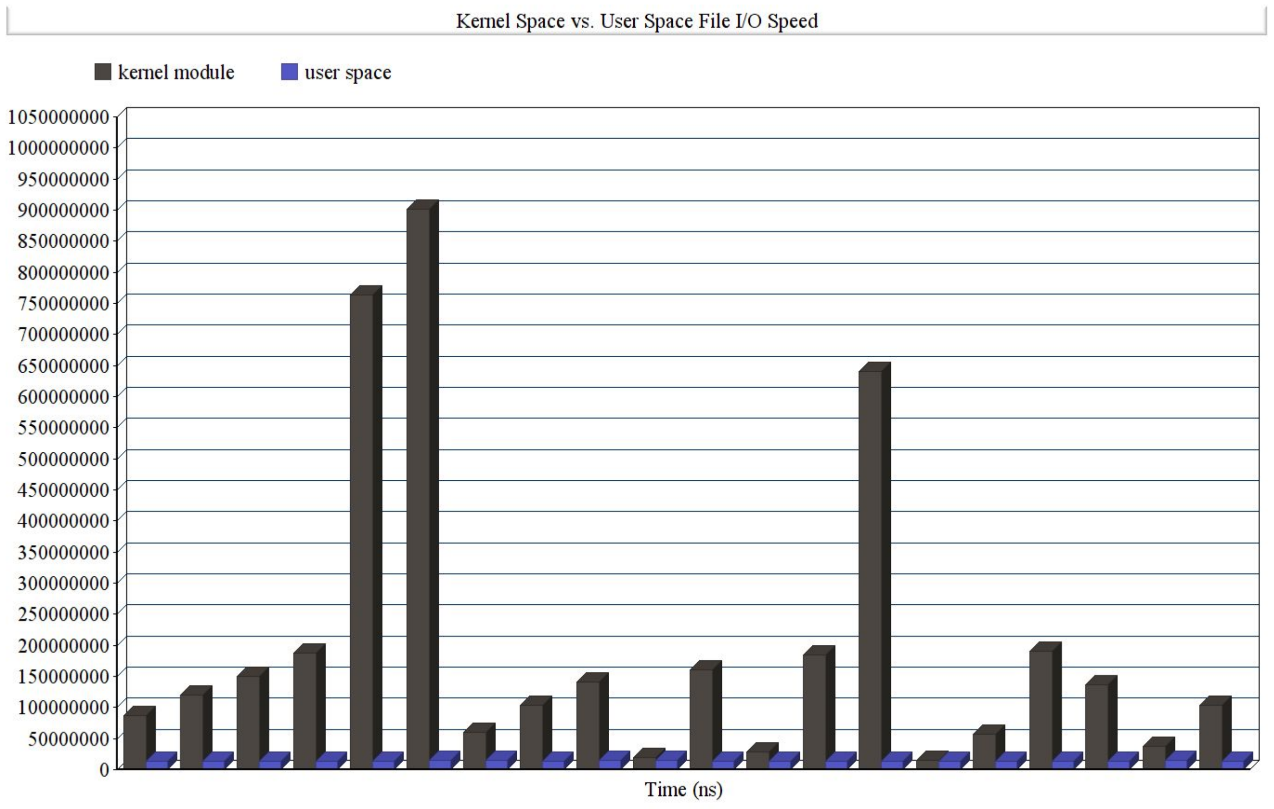
Results

Below are the times, in nanoseconds, for each of the kernel module's and c program's twenty runs.

	Kernel (ns)	User (ns)
Run #1	87051646	13969000
Run #2	119886682	13486000
Run #3	149495455	14147000
Run #4	187268744	13540000
Run #5	763098968	13993000
Run #6	901885240	14309000
Run #7	60127599	14511000
Run #8	103779854	13946000
Run #9	140250096	14625000
Run #10	19235241	14483000
Run #11	160647901	13599000
Run #12	28122113	13310000
Run #13	184108458	13655000
Run #14	640091448	13851000

Run #15	15512043	13795000
Run #16	57033044	13529000
Run #17	190512321	14140000
Run #18	136112710	14068000
Run #19	37530256	14737000
Run #20	104020559	14054000

And a graph showing the data:



Analysis

As you can see by the graph above, file I/O in a kernel module is much slower than in a simple c program. The execution time in the kernel can also vary dramatically from run to run, where the c execution time was more or less steady. This was absolutely not the result I expected to see. After having reviewed my code and methodology, and rerun the tests to the same result, the problem appears to be out of my hands. I have a number of ideas, however, as to what might be causing the surprising results.

First, it could be the loop. The c program was compiled using the latest version of gcc, whereas the kernel module had to be compiled with make, kbuild, and gcc. Perhaps there is some optimization happening under the hood with the c program that greatly speeds up the execution time, which is not enabled in the kernel module because of how the kbuild build system generates the linker flags.

Another guess is that I'm using the wrong functions, or that I'm using the functions incorrectly. It was an adventure to get the code working and the module compiling in the first place, as documentation for doing something so unnecessary and unsafe as file I/O in a kernel module is nearly non-existent. At first I tried basing my code on an article from the *Linux Journal* magazine, but that turned out to be incorrect to the point that it wouldn't even compile. I finally developed my working kernel module from piecing together several Stack Overflow answers with the oftentimes inaccessible and difficult to navigate documentation on the official linux kernel website, *www.kernel.org.* So it is safe to say that there may be another, faster way of implementing the file I/O besides the functions I chose.

Finally, my last guess is that the *vfs_write()* function may just be poorly optimized or poorly implemented, and is the main cause of the longer execution time. Because file i/o is not one of the more important parts of the standard library for kernel modules, it would make sense that it might have been given a less than perfect implementation letting the developers move on to more important things. I doubt the author of *vfs_write()* had performance at the top of their list when they wrote the function.

Conclusion

This experiment produced surprising results. I makes sense for file I/O operations to be faster within the kernel space, but that is clearly not the case. Whether this is due to the implementation of the I/O functions in the kernel header or something else, it is evident that the kernel space offers no performance benefits over the user space. Perhaps there is a method of utilizing the kernel space and achieving faster file I/O in a kernel module, but with the current state of documentation for kernel module development, it would take someone with an extensive background in kernel development to even know how to do it.