, Summer 2024

SFS ... a Multithreaded Filesystem

...gned: July 21

	Due Date	Max. Grace Days	Last Date	Weight in Course
Final	Aug 2	0	Aug 2	4%

1 Introduction

A *file system* is a data structure that the operating system uses to store, maintain, and refer to meaningful data on the disk, and conversely to derive usable information from the raw data on the disk. File systems allow programs to interact with data as abstract files, leaving cumbersome work of putting that data onto the physical disk to lower level calls. File systems are also simple but powerful tools for organization (with folders, linking, etc), and security (with permissions). In this lab, we provide you with a very basic file system, which does not implement folders, linking, or permissions. In the Shark File System, as distributed in this lab, you may only create and delete files, and read or write to them.

The first portion of this lab deals with the file system capabilities. We ask you to implement a few functions that introduce new abilities of the file system, allowing users to see and augment the position of file descriptors, and to change file names with a single call.

The next portion of this lab asks you to improve the file system's performance and safety. Currently, there is nothing to prevent two simultaneous calls to the file system from overwriting or otherwise conflicting with each other. We would like for you to prevent conflicts like these from happening, while still allowing for concurrent and fast access to the file system for separate users.

A testing environment is provided in `sfs-tester`, which can interpret commands and traces, utilizing the Shark File System.

2 Logistics

All handins are electronic. After creating you GitHub Classroom repository using "Download handout" on the sfslab Autolab page, you can clone the repository on a Shark machine with the command

```
$ git clone https://github.com/cmu15213-m24/sfslab513-m24-<YOUR USERNAME>.git
```

or

```
$ git clone git@github.com:cmu15213-m24/sfslab513-m24-<YOUR USERNAME>.git
```

3 Overview

Your first item of busin[illegible] sfs_getpos function. This function should just return the position of a file descri[illegible] should be relatively straightforward, but is a good warm-up to familiarize yourself [illegible]debase before moving on to the rest of the assignment. The principal files of interes[illegible]s-disk.h, and sfs-api.h.

Next, you should imple[illegible]ction. This function should move the file descriptor's position forwards or backwards [illegible] the argument. This will require not just accessing data stored in the file system structure like you did for sfs_getpos, but changing it as well.

After that, only sfs_rename remains unimplemented. This will also require changing data stored in the file system structure.

Once you have implemented these three functions, it is time to move on to implementing concurrency. We suggest approaching this in three gradations.

The first order of business is to make the system thread-safe in a very coarse way, only trying to ensure correctness of accesses. This can be done by simply locking the entire file system for any kind of access, and then unlocking once the request has been satisfied.

After making the file system thread-safe, you can start to think of ways to increase concurrent accesses. There are two aspects to this: reading and writing, and separate files.

When reading, no data is being changed besides the current file descriptor's position, so multiple file descriptors may read at the same time without any concern for overwriting each other, whereas writing requires solitary access.

With separate files, accesses should not ever overlap, since the files have to be separate to start with. So, reading or writing on one file need not prevent someone else from doing the same with another file.

Which aspect you start with is up to you, but we would like for you to try to implement both. If you can think of ways to improve concurrency even more, please go further!

4 The sfs Specification

You should only need to edit the sfs-disk.c file; however, some designs may need to modify the sfs-disk.h. The current top-level functions implemented in sfs-disk.c accessible via sfs-api.h are:

- `int sfs_format(const char *diskName, size_t diskSize)` will create an SFS disk image on file `diskName`, creating it if it does not exist, allocating `diskSize` bytes for the image, and then mounts this disk to be used for all subsequent `sfs-disk` file routines.
- `int sfs_mount(const char *diskName)` will mount the disk image, loading the state of a previously used file system.
- `int sfs_unmount(void)` will unmount the current disk image. If any files are open, this will fail.
- `int sfs_open(const char *fileName)` will open the file with name `fileName`, creating it if it exists, and returning a file descriptor.
- `void sfs_close(int fd)` will close the given file descriptor. This call can't fail, it can only be inert for a file descriptor that doesn't exist.

- `ssize_t sfs_write(int fd, const char *buf, size_t len)` will write up to `len` bytes of the string `buf` to the file descriptor given. On success, it returns the number of bytes written. This call updates the posit[illegible]

- `ssize_t sfs_r`[illegible]`buf, size_t len)` will read up to `len` bytes from the given file descriptor, pl[illegible] success, it returns the number of bytes read. This call updates the position of `f`[illegible]

- `int sfs_remov`[illegible]`)` will attempt to remove the file `name`.

- `int sfs_list(sfs_list_cookie *cookie, char filename_out[], size_t filename_space)` will iteratively place file names into `filename_out[]` with each call, until there are no more files. It returns 0 upon successful retrieval, 1 when it has no more filenames to write out, and possibly an error code. [untested?]

There are three functions in `sfs-disk.c` that are currently unimplemented. It is your first order of business to implement these functions.

- `ssize_t sfs_getpos(int fd)` should return the position of the file descriptor given, in bytes. Specifically, it should return the index of the byte we are on, where the first byte of the file is `0`, the second is `1`, and so on until the end of the file.

- `ssize_t sfs_seek(int fd, ssize_t delta)` should move `fd`'s position by `delta`. If given the output `loc` from an earlier `getpos` call, `sfs_seek(fd, loc - sfs_getpos(fd))` should return `fd` to the position where the file descriptor was at that earlier call. `seek` should track with reads and writes to the file descriptor, according to how many bytes have been read or written. If the requested motion would put `fd` in a negative position, it should stop at 0. If the requested motion moves past the end of the file, it should stop at the end of the file.

- `int sfs_rename(const char *old_name, const char *new_name)` should rename the file of name `old_name` to `new_name`. If `new_name` exists, then delete and replace it.

For more details, such as expected return values and error codes, refer to the specifications given in `sfs-api.h`. In addition to `sfs-disk.c` and `sfs-api.h`, `sfs-disk.h` will also be worth reading and referencing as it defines some of the high-level design choices of the file system.

5 Testing

5.1 Driver

The Python script `driver` will do a number of things.

1. First, it will run all traces given in the `traces` folder for correctness. These traces must start with a capital letter, followed by a two digit number and a dash, and must end with `.lua` (You may name traces more descriptively between the dash and `.lua`). It will automatically sort them into groups by the first letter of the filename, and output totals for each group.

2. If a trace fails, the driver will print out the error that `sfs-tester` gave and the line in the trace where the error occurred.

3. If all traces pass, the driver will run again, dynamically analyzing for races, and testing for performance. Currently, this is only done for the C traces. Passing in the -P argument will skip the first round of checks and do th[illegible]se tests are done using the instrumentation described in the Grading section [illegible]
4. The -A argument [illegible]e an Autolab-style autograder JSON output as the last line of output.

There are also options f[illegible]grams are used to test, to check, and where to look for traces and put disk images, but our default options for these should be sufficient for most purposes.

5.2 Using `sfs-tester`

We provide you with the file `sfs-tester.c` which, once compiled, becomes the binary `sfs-tester`. This executable can act as a Lua[1] interpreter, or it can take in a Lua file as an argument to run. It has been augmented from standard Lua in three ways:

- Our sfs-api.h functions are available,
- A threading library has been added, and
- Many standard Lua library functions have been disabled (primarily IO functions, and the native threading methods)

When you wish to test your file system, all you need to run is `make` to recompile the `sfs-tester` binary.

If you are unfamiliar with Lua, don't panic! We have provided a handful of traces that you can use or modify, or you can write your own by following a few simple guidelines from the Appendix.

Because `sfs-tester` is essentially a Lua interpreter, this is only the most basic way to use it to interact with the file system. Feel free to utilize the Lua language more extensively, such as writing functions. You may also take inspiration from any of the provided traces in constructing your own traces.

We provide three groups of trace files. The **A** traces do simple feature tests of the file system, particularly the functions that we ask you to implement. The **C** traces perform concurrent accesses, checking for correctness. Every **C** trace has a corresponding **B** trace of the same number, which performs the same operations sequentially, leaving the file system in the same state.

6 Grading

"Grading" on Autolab is done using the driver described earlier. Your Autolab "score" is your performance score, or zero if there is a correctness issue in an earlier trace. Autolab will also display the number of traces that pass for the *A*, *B*, and *C* categories.

As there are 5 A traces, 3 B traces, and 3 C traces, the correctness part of your grade totals 11 points. An additional 10 points are available from the measurement of performance. And a final 4 points from a style review of your submission. The style points will predominantly come from you extending the documentation of the code base based on your specific changes.

[1] Lua is a scripting language similar but unrelated to Python.

```
CT_MEM: 16630869196     15860
CT_START: 1701019792.908
CT_END: 1701019792
CT_COMP: 170101979
CT_LIMIT: 0.000
Total Contexts: 6
Total Uncomp Writt
Max Buffers Alloc:
Max RSS: 3336
```

Figure 1: Example Diagnostic Messages

6.1 Synchronization and Data Races

As you successfully implement proper synchronization in your file system, the driver can specifically test the correctness of the generated disk. Then using the instrumentation, the driver will run **raceTool** to detect whether an implementation has any data races. The tool does this by computing whether there is an explicit ordering constraint (such as synchronization) between writes or writes and reads to the same memory location.

6.2 Performance

A separate analysis will run on taskgraph from the instrumented file system to determine an estimate of the performance and parallelism for the implementation. The tool uses the number of operations and memory accesses, as well as the time required for the synchronization used to compute a modeled time for the implementation. This model is again used as Autolab is single threaded.

7 Appendix: Instrumentation

The **sfs-tester-ct** binary is built with additional instrumentation, using Contech[2]. This instrumentation generates a trace file (filename is set using the `CONTECH_FE_FILE` environment variable). The instrumentation also prints several diagnostic messages, see Figure 1. The one of immediate interest is **Total Contexts: 6**, which indicates how many threads were traced.

7.1 trace2taskgraph

The trace is then converted to a taskgraph. This binary format stores the execution of a program along with the explicit ordering information known primarily from thread creation and synchronization. These graphs can be analzyed to detect possible data races, an estimate of maximum parallelism, and many other uses.

When the tool runs correctly, the output should appear similar to Figure 2. Particularly, that this output should end with the `MIDDLE_END`.

Each line that begins with `MIDDLE_` is a time giving progress through the different steps required to convert a trace into a taskgraph. As the trace records concurrent events from different contexts, the conversion has to queue events from the trace and reconstruct the ordering constraints provided by synchronization.

[2] https://bprail.github.io/contech/

```
MIDDLE_START: 1701147105.359
Event Version set: 9    Basic Block table: 1170
Middle Space Time:
MIDDLE_QUEUE: 1701
MIDDLE_DEQUE: 1701
MIDDLE_TASK: 17011
Writing index for
Wrote 8 tasks to i
Tasks Received: 8
Tasks Written: 8
Tasks Remaining: 0
MIDDLE_END: 1701147105.370
```


Figure 2: Trace to Taskgraph Sample Output

7.2 raceTool

The **raceTool** reviews the taskgraph collected from the correctness tests by checking every memory location accessed by any part of the execution. It does this as it follows a topological sort of the execution graph. When it finds a memory location accessed that is previously accessed, it checks whether the execution had an ordering constraint between these memory accesses.

For the purposes of this lab, the two key ordering constraints are that accesses within the same thread are considered to be ordered and accesses between different threads need to have a transitive set of synchronization objects between the accesses. This means that the tool will accept execution if there are any number of synchronization objects (mutexes, semaphores, etc) that are used between common memory accesses.

There are other analysis approaches that can attempt to identify races from static code analysis; however, those are outside the scope of this assignment.

7.3 sfsperf

The second analysis of the taskgraph comes by checking the execution for its potential parallelism. The tool assigns time to the different discrete tasks from the original execution and then identifying each task that could have executed concurrently, given the sychronization used by the program. Thus we have the **work** given by the total time for the different tasks, as well as the longest sequence of tasks that have dependencies on each other, termed the critical path or **span**.

Implementations are then scored based on their ratio of work to span. To acheive a good ratio, implementations will need to minimize their span. This is accomplished by reducing the use of common sychronization objects so that tasks can execute concurrently, as well as minimizing the operations within critical sections to reduce the length of the critical path.

7.4 taskgraphViz

The taskgraph file is transformed into a visual flowchart tracing the progression of threads and synchronization points. This tool is provided to help students understand how the file system was observed executing.

8 Appendix: LUA

First, you will always w[illegible]ve a disk up and running for your test case to use. This should be done with our sfs\[illegible]diskSize) function. So, your trace file should start with a call of the form

```
disk.format("D[illegible]kSize)
```

diskSize must be a mu[illegible]m page size. Diskname.sfs need not end with .sfs, but it is our preferred conventio[illegible]ns disk.mount("Diskname.sfs") and disk.unmount() for using preformatted disk files.

After loading a disk, you can run any of the API functions that interact with files: (Functions defined in sfs-api.h as sfs_function are accessible to the test interpreter as disk.function.)

- disk.open("filename") will open the file with name filename, returning the file descriptor.
- disk.close(fd) will close the given file descriptor. This call can't fail, it can only be inert for a file descriptor that doesn't exist.
- disk.write(fd, buffer) will write the string buffer to the file descriptor given. On success, it returns the number of bytes written.
- disk.read(fd, maxbytes) will read up to maxbytes bytes from the given file descriptor, and returns the string. It does not take in a buffer location, as that is taken care of within the API function.
- disk.getPos(fd) will return the current position of the file.
- disk.seek(fd, delta) will call seek(fd, delta), and return the new position on success.
- disk.remove("filename") will attempt to remove the file with name filename.
- disk.list() will return an array of strings, each being the name of a file presently on the disk.

All of these functions, except for disk.close(fd) which cannot fail, will return a "failure tuple" in the event of some error. A failure tuple has three elements: The first one is always fail, followed by a string describing the error briefly, and then the integer error code. Unlike in C, 0 is not a false value in Lua, so checking that the result is "false" will tell you when an error has occurred. See the function check in trace A00 for a practical demonstration of this.

The three student-implemented functions, sfs_getpos, sfs_seek, sfs_rename are all accessible with the API as disk.getPos, disk.seek, disk.rename, with the same arguments as they take in sfs-disk.c, and the same return values (besides the possibility of a failure tuple).

Lua is dynamically typed, so you shouldn't try to declare variables with any type information, but you should declare them with the local keyword. For example, We can save a file descriptor for the file writeup.txt in fd1 with the line

```
local fd1 = disk.open("writeup.txt")
```

For checking correctness, there are two primary options: printing and asserting. Both are straightforward. If you want to print out a value, you can simply enclose it in a print(...) call. If you want to assert a boolean expression, you can likewise enclose it in a assert(...) call, and if it fails the interpreter will tell you on which line the assertion has failed. Strings in Lua can be directly compared with ==, which makes comparing outputs from disk.read straightforward.

9 FAQ

9.1 No space left o

Each trace sizes the 'dis rect execution. If your implementation has an error or a race, it may end up using add by causing an operation in the trace to fail.