SOFE 3950U / CSCI 3020U (Operating Systems)
Simulated UNIX File System Project
Instructor: Dr. Kamran Sartipi

Acknowledgement: this project was originally designed at the University of Waterloo.

Your project is to implement a simple file system using C or Java language. The final version of your program must compile and execute in the Linux environment. You will be given code to simulate access to a block-oriented storage device. You are to implement a file system that will reside on the simulated device. The interface and functionality of the file system are described later in this document. Your grade will be based on correctness of execution and on the quality and style of your code and documentation. At the end of this project, you will find a list of items you are expected to submit for grading.

The Block I/O System

Two routines are available to simulate access to a block-oriented storage device, e.g., a disk. The simulated device holds 512 blocks, numbered from 0 to 511. Each block holds 128 bytes. A routine called *put_block* copies 128 bytes of data from memory to one of the device blocks. The first argument to *put_block* specifies which device block to copy to. The second is a pointer to the data in memory. A routine called *get_block* copies the contents of a specified device block into memory at a specified address. Like *put-block*, it takes two arguments, one to specify the disk block and the second to specify a memory address. The *get_block* routine does not allocate memory, so the memory pointer that it is passed should point to a least 128 bytes of space that your program has allocated. Code for the *get_block* and *put_block* routines is available in the file *blockio.c,* which you should compile and link with your file system code. There is also a file *blockio.h* containing function declarations that can be included in your programs that use the block I/O functions.

You will find that these functions store the simulated device's data in a file called "*simdisk.data*". If a file with that name exists in the current directory (from a previous run of your program), the functions will use it. Otherwise, a new one will be created. When a simulated disk is first created, all of the bytes of all of the blocks of the disk are initialized to zero. You must use these functions as given with no modifications. Furthermore, your file system should use only these functions to store and retrieve data. You should not use any Linux file system calls (e.g., read(), write(), open()) in your project.

The File System

You are to implement a file system with a UNIX-like hierarchical directory structure. When a file is opened, your system should return a "file descriptor" to its caller. The file descriptor will be used, as discussed in class, to identify the open file during any subsequent operations on it. Each sfs_open() call should result in a new file descriptor, even if the file is already opened.

File Types

Your system should support two types of files: "regular files" and "directories". As discussed in class, directories are used to implement hierarchical file names. Your system should not allow the sfs_writer() or sfs_read() functions to be used on directory files. Directories are updated by the file system as necessary, e.g., when a new file is created within the directory. Directories are read using the special sfs_readdir() function.

File Names

Your system should implement a hierarchical UNIX-like file-naming scheme. The pathname "/" should refer to the root directory of the file system, and " /" should also be used to separate the components of a path name. All pathnames used as arguments to file system calls must be absolute, i.e., they must be specified

from the root directory. Thus, "/foo/bar/zam" is a legitimate pathname, but "bar/zam" is not. Each component of a pathname, e.g., "foo" or "zam", must be limited by your system to at most six characters.

Implementation Requirements
Each file must be assigned a unique internal identification number, like a UNIX i-number. Your system should use a hierarchical directory scheme for pathname-to-file-number translation. Each directory file should consist of a number of directory entries. Each directory entry should include a pathname component. Users can read the pathname components from a directory using the sfs_readdir function defined below.

File System Limits
Your file system must meet the following criteria. It must be capable of supporting at least 4 open files at one time. It must support individual files (including directories) that are up to 512 bytes long. It must allow at least 64 files to exist in the file system at one time (provided that they all fit on the disk). These are minimum requirements. Your system may be less restrictive, e.g., it may permit files to grow larger than 512 bytes. It may not be more restrictive. As noted above, the length of a pathname component must be limited to six characters. This is not a flexible criterion. Do not allow pathname components longer than six characters. This restriction is necessary so that the amount of data returned by sfs_readdir can be defined in the file system interface.

File System Interface
Your file system's interface should include the functions described below. In these descriptions, the *fd* argument is a file descriptor, returned by a previous call to *sfs_open.*

*sfs_open(char *pathname):* Open the file specified by pathname. It is an error for the file not to exist. If a file is successfully opened, a file descriptor should be returned to the caller. A file descriptor is a non-negative integer value.

*sfs_read(int fd, int start, int length, char *mem_pointer)*: Copy *length* bytes of data from a regular file to the memory location specified by *mem_pointer*. The parameter *start* gives the offset of the first byte in the file that should be copied. So, if start is 10 and length is 5, then bytes 10 through 14 of the file should be copied into memory. (Assume a zero-based count, i.e., the first byte in a file is called byte 0). If a read request cannot be fully satisfied because the file is not long enough, no data should be copied from the file, and an error value should be returned.

*sfs_write(int fd, int start, int length, char *mem_pointer):* Copy *length* bytes of data from the memory location specified by *mem_pointer* to the specified file. The parameter *start* gives the offset of the first byte in the file that should be copied to. So, if start is 10 and length is 5, then five bytes from memory should be used to "overwrite" bytes 10 through 14 in the file. Alternatively, the value of *start* may be set to -1. This indicates that the specified number of bytes should be "appended" to (added to the end of) the file. **Appending (setting *start* to -1) is the only allowable way to increase the length of a file**. This means that if start is not -1, then it is an error for (*start* + *length* -1) to be greater than the current length of the file.

*sfs_readdir(int fd, char *mem_pointer):* This call is used to read the file name components from a directory file. The first time *sfs_readdir* is called, the first file name component (a string) in the directory should be placed into memory at the location pointed to by *mem_pointer*. The function should return a positive value to indicate that a name component has been retrieved. Each successive call to *sfs_readdir* should place the next name component from the directory into the buffer, and should return a positive value. When all names have been returned, *sfs_readdir* should place nothing in the buffer, and should return a value of zero to indicate that the directory has been completely scanned.

sfs_close(int fd): Indicates that the specified file descriptor is no longer needed.

*sfs_create(char *pathname, int type):* If there is not already a file with name *pathname*, create one. If the specified name is already in use, return an error. The parameter *type* is an integer which should have value zero or one. If zero, a regular file should be created. If one, a directory file should be created. All components of pathname (except the last) must already exist, i.e., the total number of files in the file system should increase by exactly one when this command is executed without error.

*sfs_delete(char *pathname):* Delete the specified file or directory, if it exists. Directories must be empty to be deleted, i.e., the total number of files in the file system should decrease by exactly one when this command is executed without error. This call should return an error if pathname specifies a directory that is not empty.

*sfs_getsize(char *pathname)*: If the specified file is a regular file, this function should return the number of bytes in the file. If it is a directory file, this function should return the number of directory entries in the file.

*sfs_gettype(char *pathname):* This function should return the value zero if the specified file is a regular file. It should return the value one if the file is a directory.

sfs_initialize(int erase): The *sfs_initialize* function must be called before any other file system functions are called. (In addition, it may be called at any other time.) This function may be used to perform any initialization required by your file system. (See "Simulating Failures", below.) Normally, the value of erase will be zero. If the value of erase is one, this indicates that any existing file system on the simulated disk is to be destroyed, and that a brand new file system should be created. Any existing files on the device are destroyed by this call if the value of erase is set to one. A new file system should consist of a single, empty root directory and no other directories or regular files.

To simplify your implementation, you may assume that the value of the *length* and *start* parameters to the *sfs_read* and *sfs_write* calls will always satisfy the following formula:

$$\left\lfloor \frac{\mathbf{start}}{\mathbf{128}} \right\rfloor = \left\lfloor \frac{\mathbf{start + length}}{\mathbf{128}} \right\rfloor$$

In other words, no read or write operation will use data more than one device block. All of the file system calls should return an integer value. In all cases, a negative value should be used to indicate an error. Unless otherwise specified, a strictly positive value should be used to indicate successful execution. You should define a set of (negative) error codes to be used by functions that cannot execute properly. The return code should indicate the source of the problem. For example, the *sfs_create* function may fail because the specified file name is already in use, or because there is no room on the simulated disk for additional files. Each type of error should have its own return code.

Testing Your File System

This project requires you to implement a set of file system interface functions. To test your code, you must also write one or more test programs. Such programs provide a *main()* routine and call the file system functions. Do not submit your test programs along with your file system implementation. To get you started with this, and to provide you with an illustration of how the file system functions may be used, a simple interactive test program will be made available to you in the file *sfstest.c* that you should put it into your working directory. To use the interactive test program, you will need to compile and link it with your file system implementation. You may use a command similar to the following one to do this:

```
gcc -o sfstest sfstest.c your-source-files-here blockio.c
```

This should produce an executable file called "*sfstest*" which you may use to test your file system code. Look at *sfstest.c* to get an idea of how to use the interactive test program.

Simulating Failures

File system data resides permanently on disk because disk-resident data survives failures such as power outages. We will simulate failures by terminating the process in which your file system is running. The easiest way to terminate a process is to interrupt it, e.g., to type "control-C" at the keyboard.

Your file system should be able to survive any failure that occurs while there is no file system function executing. Survival means that any changes that have been made to the file system should not be lost because of the failure. In particular when running the interactive test program, it should be possible to interrupt the program any time it prompts for a new file system request to execute. If the program is then restarted, the file system should include all changes (new files, deleted files, written data, etc.) that were made before the program was interrupted. Files should never be lost unless *sfs_initialize* is used to start a new file system on the simulated disk.

After the program is restarted, *sfs_initialize* will be called before any other file system functions. When *sfs_initialize* is called, your code must determine whether a file system already exists on the simulated disk from a previous run of the program. You must determine how to do this. You should take advantage of the fact that the entire disk is initialized to Zero when it is first created. If no file system exists on the disk, or if one already exists but the erase parameter is set to one, your code should create a new, empty file system on the disk, as was described for *sfs_initialize*.

Deliverables
For more details and due dates refer to the Laboratory Link in the Blackboard System

You should submit the following items (as a single zipped file) through Blackboard System by the project due dates.

1) File system design document [30 Marks]:
This document provides an overview of the design of your file system including:

- Definitions of major data structures in your simulated Unix File System, with description of how they are used to implement the required file operations (e.g., sfs-create(…)).
- Description of how file system information is laid out on the simulated disk.
- Design of algorithms (in Pseudocode) of the above file operations.

2) Source code for simple file manipulation [25 Marks]:
Implement a single-level directory structure, as described in Subsection 11.3.3, with operations on "regular files" such as: create, delete, open, close, read, and write to the files.

3) Source code, project presentation, and evaluation of the complete file system [45 Marks]:

- Implement a "hierarchical directory structure" with all operations described in the "File System Interface" section of this document. Submit the source code of the complete file system. Your code should be neat, clear and well documented. A makefile will be available for you to compile your code. It is important that your file system interface conform exactly to the project specifications.
- Each group will present different features of their simulated Unix File System to the course TAs in the lab. The TAs will ask questions about the design document, source code, and the implemented system. The mark is based on the participation in the discussions, logic for the design, and the correctness of the simulated file system.

NOTE: Cheating Policy

The work submitted by each group must be the work of that group only. It is not permissible to share code, documentation, or design effort. You may discuss the project in general terms and may discuss the public aspects of the project (such as this handout). However, the projects must be designed, implemented and documented independently by each group. Note that it is the responsibility of each student to ensure that his or her on-line code and documentation are protected from general access. The Linux command

chmod 600 filename

(use 700 for executable files or directories) will set a file's access permissions so that only the owner of the file may read or write (or execute) it. The penalty for cheating is a grade of -100% on the project or exam.