THE UNIVERSITY OF WESTERN AUSTRALIA

Department of Computer Science and Software Engineering

CITS2002 Systems Programming

Project 2 2019. See also: Project 2 clarifications

Project Description

Modern operating systems support programs to archive the contents of file-systems. These programs typically support their own specific file-formats (examples include the *tar* and *zip* formats), or store a whole file-system image inside a single file on your operating system (examples include the *ISO* and Apple's *dmg* disk-image formats).

Whereas the *zip* archive format attempts to reduce its space requirements by compressing its contents, an alternate approach termed single-instance storage attempts to reduce space requirements by identifying multiple copies of the same content, and replacing it by a single shared copy. Thus, if multiple copies of the same file are archived, possibly from different directories, then only a single instance of the files' content will appear in the archive. Storing files' contents only once, can also reduce *internal-fragmentation*.

The **goal** of this project is to write a *user-level library* in C99, named *libsifs*, which supports the archiving of files within a simple file-system contained within a single fixed-size file, termed a *volume*.

The file-system is notable in that it stores each file (its contents) only once - if multiple copies of the same file are stored within the file-system, each file's contents will be stored only once. This provides the potential to seemingly store more data in the file-system than what appears possible.

Successful completion of the project will develop your understanding of some advanced features of the C99 programming language, and your understanding of how an operating system can support a file-system.

Download, understand, and then extend these starting files: sifs-files.zip

---

The Single Instance File System (SIFS) volume format

Our SIFS volumes are of a fixed size - once created they can neither grow nor shrink. Similarly, files on a volume are *immutable* - once a file has been added to a volume it can neither grow nor shrink, though it may be deleted. Volumes consist of three sections, whose concrete implementation is defined in the provided C99 header file *sifs-internal.h* :

1. The *volume header*, consisting of two non-negative integers. The first integer stores the number of blocks in the volume that may be used to store directory information, file information, or the contents of files. The second integer stores the size, in bytes, of each block, typically 1KB or 4KB.

2. The *volume bitmap*, records how each block on the volume is being used. While the name *volume bitmap* suggests that each element occupies only a single bit, the frequently-used name is only historic. A single 1-byte character is used to record each block's use, represented as one of four distinct values.

3. Each *block* stores either a *directory-block*, a *file-block*, a *data-block* (holding files' contents), or is *unused*.

   - *Directory-blocks* have a fixed format.
     Each includes the name of the directory and its modification time; the number of entries in the directory (either subdirectories or files); and, for each entry, the block number (blockID) of each subdirectory or file.

   - *File-blocks* have a fixed format.
     Each includes an array of characters storing the *MD5 cryptographic digest* (a summary) of the files' contents; the common length (in bytes) of every file represented by that *file-block*; the block number (blockID) of the files' first *data-block*; the number of files having the same contents; and an array of each such file's name and its modification time.

   - *Data-blocks* have no fixed format.
     They hold only files' contents, and no characters in a *data-block* have any special significance. A *first-fit*, *contiguous data-block allocation scheme* is employed - see Lecture-18. A new file can only be added to the volume if there are sufficient contiguous unused blocks to support the file's length, or if other file(s) with the same contents have already been stored.

The SIFS Application Programming Interface (API)

User-level programs do not directly interact with the contents of a SIFS volume. Instead, they call functions from your SIFS library to make and initialise new volumes; make or remove directories; list the contents of directories; and add, read, or remove files from directories. Without exposing the concrete implementation of the file-system, the library provides an *Application Programming Interface* (API) to the file-system.

Although the functions in your SIFS library execute in the same address space as your own programs, you can consider the functions as if they are new system-calls provided to support a new type of file-system on an existing operating system. Many of the library's functions need to perform the same actions, such as opening a volume, and so additional "hidden" functions within the library will be necessary.

The prototypes of the required library functions, declared in provided C99 header file *sifs.h*, are:

```
// make a new volume
int SIFS_mkvolume(const char *volumename, size_t blocksize, uint32_t nblocks);
```

```
// make a new directory within an existing volume
int SIFS_mkdir(const char *volumename, const char *dirname);

// remove an existing directory from an existing volume
int SIFS_rmdir(const char *volumename, const char *dirname);

// add a copy of a new file to an existing volume
int SIFS_writefile(const char *volumename, const char *pathname, void *data, size_t nbytes);

// read the contents of an existing file from an existing volume
int SIFS_readfile(const char *volumename, const char *pathname, void **data, size_t *nbytes);

// remove an existing file from an existing volume
int SIFS_rmfile(const char *volumename, const char *pathname);

// get information about a requested directory
int SIFS_dirinfo(const char *volumename, const char *pathname, char ***entrynames, uint32_t *nentries, time_t *modtime);

// get information about a requested file
int SIFS_fileinfo(const char *volumename, const char *pathname, size_t *length, time_t *modtime);
```

Each of the library's functions returns 0 on success, and 1 on failure. Each failed library call also sets the global integer variable *SIFS_errno* to describe the cause of the error. All possible values of *SIFS_errno* are defined in provided C99 header file *sifs.h*.

Examples of use

This project requires a *library* of functions, named *libsifs.a* , to be developed. The library does not contain a *main()* function and so, by itelf, it is not an executable program. Instead, other programs requiring access to a SIFS volume and its contents will be *linked* with the *libsifs.a* library and, when executed, will call the SIFS API functions and check their return value(s).

For example, imagine we have a simple program, named *sifs_put* to copy a file from the current directory to a SIFS volume. We could compile and link such a program with:

```
prompt>  cc -std=c99 -Wall -Werror -pedantic -o sifs_put sifs_put.c -lsifs -lm
```

and then execute the program with:

```
prompt>  ./sifs_put  myvolumename  myfilename
```

where the program's argument *myvolumename* provides the name of the SIFS volume (a large single file holding the archive) and *myfilename* provides the name of the (local) file to be stored on the volume. As a convenience, the *sifs_put* program could obtain the desired SIFS volume name from an *environment variable*, and then not require it on the command-line:

```
prompt>  export SIFS_VOLUME=/Users/chris/sifs-1
prompt>  ./sifs_put  myfilename
```

Remembering that, once created, a SIFS volume has a fixed size, and that its role is to store only a single copy of duplicate file content, then the following commands will store two (identical) files but not require any additional space:

```
prompt>  ./sifs_put  myfilename
prompt>  cp  myfilename  myfilename-today
prompt>  ./sifs_put  myfilename-today
```

Notes

- To develop a portable file-system format, a number of data-types are used that may be new to some students:

  uint32_t - a non-negative 32-bit integer, used to count things.
  size_t - a non-negative 64-bit integer, representing the size (in bytes) of things, as returned by the *strlen()* function.
  time_t - a 64-bit integer, representing the number of seconds since Jan 1st 1970, as returned by the *time()* function.

- API functions that receive character strings as parameters define those parameters with **const char ***
  const is a C99 keyword, short for *constant*, and is used here when a function promises not to modify what its parameter points to.

- API functions do not print anything - even if they detect an error.

- A volume's *root directory* is always identified by the first *bitmap* entry; occupies the first *directory-block*; has the empty-string as its directory *name*; and cannot be deleted.

- Each directory or file name may be up to 32 bytes long, including its terminating NULL-byte. No other character has any significance in a name. The *directory separator character*, '/', is *not* stored in any directory or file name.

- Unlike traditional Unix-style directories, directories on a SIFS volume do not contain entries for "." or ".."
  (because processes can never 'cd' into a SIFS volume, or traverse 'up' a filetree).

- Each modification time is stored in an integer (a data-type named *time_t*), as returned by Unix's *time()* function. Even though files (their contents) cannot be modified, we still refer to their *modification time* rather than their *creation time* to be consistent with the *modification time* of each *directory-block*.

- The *MD5digest()* function returns a pointer to a fixed-size block of 16 bytes, termed a *message-digest*. Two files are deemed to be *identical* if the *MD5digest()* function returns identical digests for each file. Your project **must** employ MD5 digests to determine and record if two files are identical. While all file names will be stored on the volume, the *data-blocks* of identical files should only be stored once per volume. A message-digest is not a traditional C string and, so, is not guaranteed to be end in a NULL-byte. However, the provided *sprintMD5()* function assists by formatting an MD5 digest as a 32 character string.

- **It is anticipated** that a successful project will need to use (at least) the standard C99 and POSIX functions:
  - *time()*, *ctime()*,
  - *memset()*, *memcmp()*, *memcpy()*,
  - *strlen()*, *strcpy()*, *strcmp()*,
  - *malloc()*, *free()*,
  - *stat()*, *fopen()*, *fread()*, *fwrite()*, *fseek()*, and *fclose()*.

Project requirements

1. Your library **must** be named *libsifs.a*

2. The functions in your library **must** be developed in multiple source files and **must** be compiled and linked using a *Makefile*, containing approriate variable definitions and automatic variables.

3. Your library **must** provide all of the API functions listed above. The source-code for the *SIFS_mkvolume()*, *SIFS_perror()* and *MD5* functions are provided for you.

4. Your library **must** use the provided values of *SIFS_errno* to report its own errors. For example, if requested to *delete* a file that does not exist on a volume, your library should set *SIFS_errno = SIFS_ENOENT;* and the function should return 1.

5. Your project **must** employ sound programming practices, including the use of meaningful comments, well chosen identifier names; appropriate choice of basic data-structures, data-types, and functions; and appropriate choice of control-flow constructs.

6. While it is possible for all students to gain full marks (40/40) by completing all of the standard requirements, those wishing a challenge may wish to develop an additional API function named *SIFS_defrag()*. The function will attempt to coalesce all unused blocks into a single contiguous area at the end of the volume's third section. This will possibly enable new large files to be added to a volume that previously suffered from much *external-fragmentation*. Completing this additional task will not push your final mark beyond 40/40, but you may **reclaim up to 8 marks** that you lost in the other stages of the marking.

Assessment

The project is due **11:59PM Fri 18th October**, and is worth **20% of your final mark** for CITS2002.
It will be marked out of 40. The project may be completed **individually or in teams of two** (but not teams of three).

You are **strongly** encouraged to work with someone else - this will enable you to discuss your initial design, and to assist each other to develop and debug your joint solution. Work together - do not attempt to split the project into two equal parts, and then plan to meet near the deadline to join your parts together.

20 of the possible 40 marks will come from the correctness of your solution. The remaining 20 marks will come from your programming style, including your use of meaningful comments; well chosen identifier names; appropriate choice of basic data-structures, data-types and functions; and appropriate choice of control-flow constructs.

During the marking, attention will obviously be given to the correctness of your solution. However, a correct and efficient solution should not be considered as the perfect, nor necessarily desirable, form of solution. Preference will be given to well presented, well documented solutions that use the appropriate features of the language to complete tasks in an easy to understand and easy to follow manner. That is, do not expect to receive full marks for your project simply because it works correctly. Remember, a computer program should not only convey a message to the computer, but also to other human programmers.

Your project will be marked on the computers in CSSE Lab 2.03, using the macOS environment. No allowance will be made for a program that *"works at home"* but not on CSSE Lab 2.03 computers, so be sure that your code compiles and executes correctly on these machines before you submit it.

Submission requirements

1. The deadline for the project is **11:59PM Friday 18th October (end of week 11)**.

2. Your submission will be compiled, run, and examined using the macOS platform on computers in CSSE Lab 2.03. Your submission must work as expected on this platform. While you may develop your project on other computers, excuses such as *"it worked at home, just not in the lab!"* will not be accepted.

3. Your submission's C99 source file should each begin with the C99 block comment:

```
/* CITS2002 Project 2 2019
   Name(s):             student-name1 (, student-name2)
   Student number(s):   student-number-1 (, student-number-2)
 */
```

   If working as a team, only one team member should make the team's submission.

4. **You must submit your project electronically using *cssubmit*.** You should submit all C99 source-code (*.c) and header (*.h) files and a *Makefile* that specify the steps required to compile and link your library. You can submit multiple files in one submission by first archiving them with *zip* or *tar*. The *cssubmit* facility will give you a receipt of your submission. You should print and retain this receipt in case of any dispute. Note also that the *cssubmit*

facility does not archive submissions and will simply overwrite any previous submission with your latest submission.

5. You **do not need to** submit any additional application programs or testing scripts that you used while developing your project.

6. This project is subject to UWA's Policy on Assessment - particularly §10.2 *Principles of submission and penalty for late submission*. In accordance with this policy, you may *discuss* with other students the general principles required to understand this project, but the work you submit must be the result of your own efforts. All projects will be compared using software that detects significant similarities between source code files. Students suspected of plagiarism will be interviewed and will be required to demonstrate their full understanding of their project submission.

Good luck!

Chris McDonald.