CS342 Operating Systems - Fall 2019
Project #4
SFS - Simple File System

Document Version: v1.2

Assigned: Dec 17, 2019, Tuesday.
Due date: Jan 04, 2020, Saturday, 23:55.

- *Submit through Moodle. Make sure you start submitting one day before the deadline. You can overwrite your submission as many times as you wish. Late submissions will not be accepted (no excuse; no email will be accepted).*
- *You will develop your project in Linux/C. The project can be done in groups of 2.*

1 Assignment

In this project you will implement a very simple file system. The file system will be implemented as a library (**libsimplefs.a**) and will store files in a virtual disk. The virtual disk will be a simple regular Linux file of some certain size. An application that would like to create and use files will be linked with the library. We will assume that the virtual disk will be used by one process at a time. When the process using the virtual disk terminates, then another process that will use the virtual disk can be started. With this assumption, we will not worry about race conditions.

1.1 Interface

The library will implement the following functions that can be called by an application. These functions will be implemented in a file called **simplefs**.**c**. The prototypes of these functions will be included in a header file called **simplefs.h**. In simplefs.c, you can implement and use some additional functions that will not be called by applications directly.

int **create_vdisk** (char *vdiskname, int m).
This function will be used to create a virtual disk (i.e., a simple Linux file) of certain size. The name of the Linux file is vdiskname. The parameter m is used to set the size. Size will be 2^m bytes. If success, 0 will returned; if error, -1 will be returned.

int **sfs_format** (char *vdiskname).
This function will be used to initialize/create an sfs file system on the virtual disk (high-level formatting the disk). On disk file system structures (like superblock, FAT, etc.) will be initialized as part of this call. If success, 0 will be returned. If error, -1 will be returned.

int **sfs_mount** (char *vdiskname).
This function will be used to mount the file system, i.e., to prepare the file system be used. This is a simple operation. Basically, it will open the regular Linux file (acting as the virtual disk) and obtain an integer file descriptor. Other operations

in the library will use this file descriptor. This descriptor will be a global variable in the library. If success, 0 will be returned; if error, -1 will be returned.

int **sfs_umount** ().

This function will be used to unmount the file system: flush the cached data to disk and close the virtual disk (Linux file) file descriptor. It is a simple to implement function. If success, 0 will be returned, if error, -1 will be returned.

int **sfs_create** (char *filename).

With this, an applicaton will create a new file with name filename. You library implementation of this function will use an entry in the root directory to store information about the created file, like its name, size, first data block number, etc. If success, 0 will be returned. If error, -1 will be returned.

int **sfs_open** (char *filename, int mode).

With this an application will open a file. The name of the file to open is filename. The mode paramater specifies if the file will be opened in read-only mode or in append-only mode. If 0, read-only; if 1, append-only. We can either read the file or append to it. A file can not be opened for both reading and appending at the same time. In your library you should have a open file table, and an entry in that table will be used for the opened file. The index of that entry can be returned as the return value of this function. Hence the return value will be a non-negative integer acting as a file descriptor to be used in subsequent file operations. If error, -1 will be returned.

int **sfs_getsize** (int fd).

With this an application learns the size of a file whose descriptor is fd. File must be opened first. Returns the number of data bytes in the file. A file with no data in it (no content) has size 0. If error, returns -1.

int **sfs_close** (int fd).

With this an application will close a file whose descriptor is fd. The related open file table entry should be marked as free.

int **sfs_read** (int fd, void *buf, int n).

With this, an application can read data from a file. fd is the file descriptor. buf is pointing to a memory area for which space is allocated earlier with malloc (or it can be a static array). n is the amount of data to read. Upon failure, -1 will be returned. Otherwise, number of bytes sucessfully read will be returned.

int **sfs_append** (int fd, void *buf, int n).

With this, an application can append new data to the file. The parameter fd is the file descriptor. The parameter buf is pointing to (i.e., is the address of) a static array holding the data or a dynamically allocted memory space holding the data. The parameter n is the size of the data to write (append) into the file. If error, -1 will be returned. Otherwise, the number of bytes successfully appended will be returned.

int **sfs_delete** (char *filename).

With this, an application can delete a file. The name of the file to be deleted is filename. If succesful, 0 will be returned. In case of an error, -1 will be returned.

Assume, a process can open at most 10 files simultaneously. Hence your library should have an open file table that should have 10 entries.

Below is an application showing how the functions can be used.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "simplefs.h"

#define DISKNAME "vdisk1.bin"

int main()
{
    int ret;
    int fd1, fd2, fd;
    int i;
    char buffer[1024];
    char buffer2[8] = {50, 50, 50, 50, 50, 50, 50, 50};
    int size;
    char c;

    printf ("started\n");

    // ************************************************
    // if this is the first running of app, we can
    // create a virtual disk and format it as below
    ret  = create_vdisk (DISKNAME, 24); // size = 16 MB
    if (ret != 0) {
      printf ("there was an error in creating the disk\n");
      exit(1);
    }
    ret = sfs_format (DISKNAME);
    if (ret != 0) {
      printf ("there was an error in format\n");
      exit(1);
    }
    // ************************************************

    ret = sfs_mount (DISKNAME);
    if (ret != 0) {
      printf ("could not mount \n");
      exit (1);
    }

    printf ("creating files\n");
    sfs_create ("file1.bin");
    sfs_create ("file2.bin");
    sfs_create ("file3.bin");

    fd1 = sfs_open ("file1.bin", MODE_APPEND);
    fd2 = sfs_open ("file2.bin", MODE_APPEND);
    for (i = 0; i < 10000; ++i) {
      buffer[0] =   (char) 65;
      sfs_append (fd1, (void *) buffer, 1);
    }

    for (i = 0; i < 10000; ++i) {
      buffer[0] = (char) 70;
      buffer[1] = (char) 71;
      buffer[2] = (char) 72;
```

```
      buffer[3] = (char) 73;
      sfs_append(fd2, (void *) buffer, 4);
    }

    sfs_close(fd1);
    sfs_close(fd2);

    fd = sfs_open("file3.bin", MODE_APPEND);
    for (i = 0; i < 10000; ++i) {
      memcpy (buffer, buffer2, 8); // just to show memcpy
      sfs_append(fd, (void *) buffer, 8);

    }
    sfs_close (fd);

    fd = sfs_open("file3.bin", MODE_READ);
    size = sfs_getsize (fd);
    for (i = 0; i < size; ++i) {
      sfs_read (fd, (void *) buffer, 1);
      c = (char) buffer[0];
    }
    sfs_close (fd);

    ret = sfs_umount();
}
```

1.2 File System Specification

The sfs file system will have just a single directory, i.e., root directory, so that it will be simple to implement. No subdirectories are supported. The block size is 1 KB. Block 0 (first block) will contain superblock information.

The next 7 blocks, ie., blocks 1, 2, 3, 4, 5, 6, 7, will contain the *root directory*. Fixed sized *directory entries* will be used. Directory entry size is 128 bytes. That means each disk block can hold 8 directory entries. In this way we can have at most 7x8 = 56 directory entries, hence the file system can store at most 52 files in the disk. Maximum filename is 32 characters long, incuding the null character at the end. A directory entry for a file will contain filename and some other attributes that you find necessary.

FAT (*file allocation table*) method is used to keep track of blocks allocated to files and free blocks. *FAT entry size* is 8 bytes. FAT will be stored after the root directory in the disk. That means the next 1024 disk blocks (after the root directory blocks) will store the FAT informaton. FAT will occupy 1024 disk blocks no matter how big the virtual disk is. Each disk block can store 1024/8 = 128 FAT entries. Hence, FAT will have 1024 x 128 = 128K entries. With those many entries possible in the FAT, the maximum disk size can be 128K x 1 KB = **128 MB**. After FAT, data blocks will come in the virtual disk. The rest of the disk will be data blocks. The disk should be large enough to hold at least the FAT, the root directory and the superblock. Hence minimum disk size is 1024+7+1 = 1032 disk blocks, which is slightly more than **1 MB.**

The rest is upto you to specify. For example, you will decide what to keep in a directory entry, in a FAT entry, etc.

1.3 Experimentation and Report

Do some timing experiments and write a report about the results. Try to measure how long it takes to create, read, write a file. Try various sizes. Plot results. Try to draw conclusions.

In your report, you must also write the details of your file system design; the structure of entries, etc.

2 Sample Code

Makefile

```
all: libsimplefs.a app

libsimplefs.a:    simplefs.c
      gcc -Wall -c simplefs.c
      ar -cvq libsimplefs.a simplefs.o
      ranlib libsimplefs.a

app:  app.c
      gcc -Wall -o app app.c  -L. -lsimplefs

clean:
      rm -fr *.o *.a *~ a.out app
```

simplefs.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "simplefs.h"

int vdisk_fd; // global virtual disk file descriptor
              // will be assigned with the sfs_mount call
              // any function in this file can use this.


// This function is simply used to a create a virtual disk
// (a simple Linux file including all zeros) of the specified size.
// You can call this function from an app to create a virtual disk.
// There are other ways of creating a virtual disk (a Linux file)
// of certain size.
// size = 2^m Bytes
int create_vdisk (char *vdiskfilename, int m)
{
    char command[BLOCKSIZE];
    int size;
    int num = 1;
```

```
    int count;
    size  = num << m;
    count = size / BLOCKSIZE;
    printf ("%d %d", m, size);
    sprintf (command, "dd if=/dev/zero of=%s bs=%d count=%d",
           vdiskfilename, BLOCKSIZE, count);
    printf ("executing command = %s\n", command);
    system (command);
    return (0);
}


// read block k from disk (virtual disk) into buffer block.
// size of the block is BLOCKSIZE.
// space for block must be allocated outside of this function.
// block numbers start from 0 in the virtual disk.
int read_block (void *block, int k)
{
    int n;
    int offset;

    offset = k * BLOCKSIZE;
    n = read (vdisk_fd, block, BLOCKSIZE);
    if (n != BLOCKSIZE) {
      printf ("read error\n");
      return -1;
    }
    return (0);
}

// write block k into the virtual disk.
int write_block (void *block, int k)
{
    int n;
    int offset;

    offset = k * BLOCKSIZE;
    n = write (vdisk_fd, block, BLOCKSIZE);
    if (n != BLOCKSIZE) {
      printf ("write error\n");
      return (-1);
    }
    return 0;
}



/*****************************************************************
The following functions are to be called by applications directly.
*****************************************************************/

int sfs_format (char *vdiskname)
{
    return (0);
}

int sfs_mount (char *vdiskname)
{
    // simply open the Linux file vdiskname and in this
    // way make it ready to be used for other operations.
    // vdisk_fd is global; hence other function can use it.
    vdisk_fd = open(vdiskname, O_RDWR);
    return(0);
```

```
}

int sfs_umount ()
{
    fsync (vdisk_fd);
    close (vdisk_fd);
    return (0);
}


int sfs_create(char *filename)
{
    return (0);
}


int sfs_open(char *file, int mode)
{
    return (0);
}

int sfs_close(int fd){
    return (0);
}

int sfs_getsize (int  fd)
{
    return (0);
}

int sfs_read(int fd, void *buf, int n){
    return (0);
}


int sfs_append(int fd, void *buf, int n)
{
    return (0);
}

int sfs_delete(char *filename)
{
    return (0);
}
```

3 Submission

Put all your files into a directory named with your Student Id. In a README.txt file, write your name, ID, etc. Include a Makefile. Then tar and gzip the directory. For example a student with ID 21404312 will create a directory named "21404312" and will put the files there. Then he will tar the directory (package the directory) as follows:

tar cvf 21404312.tar 21404312

Then he will gzip the tar file as follows:

gzip 21404312.tar}

In this way he will obtain a file called 21404312.tar.gz. Then he will upload this file in Moodle.

Late submission will not be accepted (no exception). A late submission will get 0 automatically (you will not be able to argue it). Make sure you make a submission one day before the deadline. You can then overwrite it.

Each group will make one submission. One of the IDs can be used. A README file must be included to give information about the group.

4 Tips and Clarifications

- Your library will be a static library (.a). See Makefile provided about how a static library can be created.
- Sample code is provided in github. You can download and use it as the starting point.
  https://github.com/korpeoglu/cs342fall2019-p4
- You should access the virtual disk in blocks (block by block). That is quite simple. Example is shown in simplefs.c.