Bachelor's in Computer Science and Engineering

Bachelor's in Computer Science and Engineering and Business Administration

Programming Assignment 2: File System

Operating Systems Design

David del Rio Astorga
Javier López Gómez

March 20, 2018

Contents

1 Introduction

The aim of this programming assignment is to develop a simplified userspace file system using the C language (ISO C11) on the Debian 9.4 operating system, whose storage will be backed by a file in the host operating system.

In this assignment, the student will:

1. Design the architecture of the file system, its control structures (e.g. i-nodes, superblock or file descriptors) and algorithms in order to comply with the requirements and features defined in this document.

2. Implement the former design in C language, and develop client programs relying on the interface of this file system to access and modify the files within.

3. Justify the design and implementation internals of the file system, in tight relation to the theoretical concepts covered in the Operating Systems Design subject. This is particularly critical for implementation dependant or undefined behaviours.

4. Build a validation plan for the resulting design against the given requirements.

5. Summarise and discuss the former points in a written report.

All of these items will be taken into consideration for the evaluation of the assignment.

2 Evaluation

The final mark of this project is obtained as follows:

- **Design** (*4 points*).
- **Validation plan** (*2 point*).
- **Implementation** (*4 points*)[1].

IMPORTANT: groups must be declared **before April 7th** in order to be considered for the continuous assessment. Please send an email to your lab teacher with the following information about the group members:

1. Full name.
2. E-mail.
3. NIA.
4. Enrolled group.

[1]The implementation will be evaluated during the lab session following submission. These results could be nullified in case plagiarism is detected afterwards (see Sec. 6)

3 Initial Code

The file *dssoo_fs.zip* contains baseline code and utilities for your convenience. Inside you will find a folder containing the following:

- **Files that CANNOT be modified**

  1. `include/blocks_cache.h`: declaration of the functions to read or write blocks from/to the disk. This includes the `bread` and `bwrite` functions to read and write, respectively, a full block from the device given its block number. It is strictly **FORBIDDEN** to access the disk in a different manner.
  2. `blocks_cache.c`: implementation of the functions to read or write blocks from/to the disk.
  3. `include/crc.h`: declaration of auxiliary functions to compute cyclic redundancy checks (CRC). This includes the `CRC16` and `CRC32` functions to compute CRCs of 16 and 32 bits, respectively, given a buffer and its length in bytes.
  4. `crc.c`: implementation of the auxiliary functions to compute CRCs.
  5. `include/filesystem.h`: declaration of the interface with the file system. This interface has to be implemented in `filesystem.c`.
  6. `create_disk.c`: creates `disk.dat` files with different sizes.
  7. `Makefile`: used by the build tool to compile the program. Use `make` to compile the program and `make clean` to remove compiled files.

- **Files to be completed by the student**

  1. `filesystem.c`: implementation of the interface to use the file system. The student must use this file as starting point in order to develop the file system requested. Any additional function developed by the student must be implemented here, and it must be exclusively internal to this file.
  2. `include/metadata.h`: definition of the structures, data types and constants defined by the student in order to implement the file system.

     Additionally, you can make use of the `bitmap_getbit()` and `bitmap_setbit()` functions to handle bit arrays of arbitrary length, and will be useful to track used blocks.

     `bitmap_getbit(bitmap_, i_)`: gets the state of the *i*-th bit in the bitmap referenced to by `bitmap_` (of type `char *`).

     `bitmap_setbit(bitmap_, i_, val_)`: sets to `val_` the state of the *i*-th bit in the bitmap referenced to by `bitmap_` (of type `char *`).

     The following listing gives an idea of their usage:

     ```
     char bitmap[2];  // array of lenth = 16 bits
     int val = bitmap_getbit(bitmap, 7);
     printf("%d\n", val); // Value of bit 7 = 0
     bitmap_setbit(bitmap, 7, 1);
     val = bitmap_getbit(bitmap, 7);
     printf("%d\n", val); // Value of bit 7 = 1
     ```

  3. `include/auxiliary.h`: definition of auxiliary functions for the core functions defined in `filesystem.h`. These functions must be implemented in `filesystem.c` along with the core functionality, and they cannot be used outside the later (i.e. this file does not extend the interface of the file system).

4. `test.c`: includes a minimal set of tests to check some features of the file system. This file should be extended with further tests created by the student to validate the features of the file system exhaustively, and check all possible errors detailed in the interface.

It is strictly **FORBIDDEN** to modify the signature of the functions in the files `blocks_cache.h`, `crc.h`, and `filesystem.h` (i.e. name, parameters, return type).

Further information on the return values of the provided functions can be found in the documentation in the header files. The student must consult this information before proceeding to the implementation of the design.

4 Specification of Functionality

The student has to design and implement from scratch a file system able to manage an emulated storage device (`disk.dat`). Besides supporting the standard operations of a file system (e.g. `mount`, `open`, `read`, `write`, etc.), the student will integrate data integrity checking mechanisms to detect data corruption of file contents. This additional functionality is key for file systems, since hardware failures and malicious corruptions are likely to occur throughout a file system's life cycle. To conduct these tasks, the student shall rely on the CRC functionality provided as part of the initial code.

The following sections define in detail the client interface for the whole requested functionality. These functions are declared in `filesystem.h`, and have to be implemented within `filesystem.c`.

4.1 Device Management

```
int mkFS(long deviceSize)
```

- **Behaviour:** generates the proper file system structure in a storage device, as designed by the student.
- **Parameters:**
  - `deviceSize` – Size of the disk to be formatted, in *bytes*.
- **Return:**
  - `0` – The execution is correct.
  - `-1` – In case of error. Trying to create a file system which exceeds the storage capacity limit of the device is considered an error.

```
int mountFS(void)
```

- **Behaviour:** this is the first FS-related operation to be executed by a client program in order to interact with the files. This function mounts the simulated device *–disk.dat–*, hence allocating and setting up all the structures and variables necessary to use the FS.
- **Parameters:** none.
- **Return:**
  - `0` – The execution is correct.
  - `-1` – In case of error.

```
int unmountFS(void)
```

- **Behaviour:** this is the last FS-related operation to be executed by a client program, as it unmounts the simulated device *disk.dat*. This function frees all the structures and variables used by the FS.

- **Parameters:** none.
- **Return:**
  - `0` – The execution is correct.
  - `-1` – In case of error.

4.2 File Management

```
int createFile(char *fileName)
```

- **Behaviour:** performs all the necessary changes in the file system in order to create a new, empty file.
- **Parameters:**
  - `fileName` – Name of the file to create.
- **Return:**
  - `0` – The execution is correct.
  - `-1` – The file cannot be created because it already exists in the file system.
  - `-2` – In case of other errors.

```
int removeFile(char *fileName)
```

- **Behaviour:** performs all the necessary changes in the file system in order to remove a file.
- **Parameters:**
  - `fileName` – Name of the file to remove.
- **Return:**
  - `0` – The execution is correct.
  - `-1` – The file cannot be removed because it does not exist in the file system.
  - `-2` – In case of other errors.

```
int openFile(char *fileName)
```

- **Behaviour:** opens an existing file in the file system and initialises its seek pointer to the beginning of the file.
- **Parameters:**
  - `fileName` – Name of the file to open.
- **Return:**
  - The file descriptor of the opened file.

    - `-1` – The file cannot be opened because it does not exist in the file system.
    - `-2` – In case of other errors.

```
int closeFile(int fileDescriptor)
```

- **Behaviour:** closes an opened file.
- **Parameters:**
    - `fileDescriptor` – File descriptor of the file to close.
- **Return:**
    - `0` – The execution is correct.
    - `-1` – In case of error.

4.3 Interacting with Files

```
int readFile(int fileDescriptor, void *buffer, int numBytes)
```

- **Behaviour:** reads as many bytes as indicated from a given file descriptor, starting from the seek pointer of the file, and placing the data into the provided buffer. This function increments the seek pointer of the file as many bytes as read from the file.
- **Parameters:**
    - `fileDescriptor` – File descriptor of the file to read from.
    - `buffer` – Buffer that will store the read data after the execution of the function.
    - `numBytes` – Number of bytes to read from the file.
- **Return:**
    - Number of bytes correctly read. Consider that if the remaining amount of bytes to be read from the file is lower than the `numBytes` parameter, the function will return the former. In particular, reading when there are no remaining bytes will result in a return value of 0.
    - `-1` – In case of error.

```
int writeFile(int fileDescriptor, void *buffer, int numBytes)
```

- **Behaviour:** modifies the file given by a file descriptor, starting from the seek pointer of the file. It stores as many bytes as indicated from a user-provided buffer. This function increments the seek pointer of the file as many bytes as written into the file. Note that in case the operation exceeds the number of data blocks initially reserved for the file, new data blocks shall be reserved without violating the filesystem limits.
- **Parameters:**
    - `fileDescriptor` – File descriptor of the file to write into.

    - `buffer` – Data to be written.
    - `numBytes` – Number of bytes to write from the buffer.

- **Return:**
    - Number of bytes correctly written. Trying to write past the maximum file size will not be considered an error, and the number of bytes written in the block will be returned. Hence, trying to write when the pointer is at EOF will return 0 bytes written.
    - `-1` – In case of error.

```
int lseekFile(int fileDescriptor, int whence, long offset)
```

- **Behaviour:** modifies the value of the file seek pointer according to a given reference and offset.
- **Parameters:**
    - `fileDescriptor` – File descriptor.
    - `whence` – Constant value acting as reference for the seek operation. This could be:
        * `FS_SEEK_CUR` – Current position of the seek pointer.
        * `FS_SEEK_BEGIN` – Beginning of the file.
        * `FS_SEEK_END` – End of the file.
    - `offset` – Number of bytes to displace the seek pointer from the `FS_SEEK_CUR` position. This value can be either positive or negative, although it will never allow positioning the seek pointer outside the limits of the file. If `whence` is set to `FS_SEEK_BEGIN` or `FS_SEEK_END`, the file pointer must be set to this position, regardless of the offset value.
- **Return:**
    - `0` – The execution is correct.
    - `-1` – In case of error.

4.4 Integrity Control

```
int checkFile(char *fileName)
```

- **Behaviour:** verifies the integrity of the data associated to a specified file.
- **Parameters:**
    - `fileName` – Name of the file to verify.
- **Return:**
    - `0` – The execution is successful and the file is correct.
    - `-1` – The execution is successful, but the file is corrupted.
    - `-2` – In case of error. Trying to execute this function when the file is opened is considered an error.

4.5 Undefined Behaviours

Any behaviour or architectural feature not specified in this document is subject to the student's judgement, thus considered a *design decision*. Since the objective of this assignment is to design a file system, design decisions have a key role in assessing the student's knowledge on the topic.

The student has to **clearly state in the report** those characteristics which are *implementation dependant* or *undefined*, i.e. out of the scope of this document, but not in conflict with this specification or the requirements defined in Sec. 5. Therefore, the student must describe these problems, ambiguities, or obscure issues, and justify the selected approach.

5 Specification of Requirements

The student must design and implement the file system to attain the functionality described in Sec. 4. In addition, the student shall comply with the requirements defined in this section. It is highly recommended that the student verifies the compliance with those.

5.1 Functional Requirements

F1 The file system shall support the following core functionality:

- **F1.1** Create a file system (function `mkFS`).
- **F1.2** Mount a file system (function `mountFS`).
- **F1.3** Unmount a file system (function `unmountFS`).
- **F1.4** Create a file within the file system (function `createFile`).
- **F1.5** Remove an existing file from the file system (function `removeFile`).
- **F1.6** Open an existing file (function `openFile`).
- **F1.7** Close an opened file (function `closeFile`).
- **F1.8** Read from an opened file (function `readFile`).
- **F1.9** Write to an opened file (function `writeFile`).
- **F1.10** Modify the position of the seek pointer (function `lseekFile`).
- **F1.11** Check the integrity an existing file (function `checkFile`).

F2 Every time a file is opened, its seek pointer will be reset to the beginning of the file.

F3 Metadata shall be updated after any write operation in order to properly reflect any modification in the file system.

F4 The file system will not implement directories.

F5 File integrity must be checked, at least, on `open` operations.

F6 The whole contents of a file could be read by means of several `read` operations.

F7 A file could be modified by means of `write` operations.

F8 As part of a `write` operation, file capacity may be extended by means of additional data blocks.

F9 The file system can be created on partitions of the device smaller than its maximum size.

5.2 Non-functional Requirements

NF1 The maximum number of files in the file system will never be higher than 40.

NF2 The maximum length of the file name will be 32 characters.

NF3 The maximum size of the file will be 1 MiB.

NF4 The file system block size will be 2048 bytes.

NF5 Metadata shall persist between `unmount` and `mount` operations.

NF6 The file system will be used on disks from 50 KiB to 10 MiB.

NF7 The size of on-disk filesystem metadata shall be minimized.

NF8 The implementation shall not waste available resources.

5.3 Implementation Requirements

I1 The submitted code has to work on the *guernika* server. You must compile and test your code in *guernika* before submitting[2].

I2 The code must compile with the flags `-Werror` and `-Wall` in order to be evaluated. Programs that do not compile will have a grade of zero.

I3 The files `blocks_cache.h`, `blocks_cache.c`, `filesystem.h`, `create_disk.c`, `Makefile`, `crc.h`, and `crc.c` cannot be modified[3].

I4 The storage device will be emulated in a single file named `disk.dat`. Using other files is forbidden.

I5 The implementation of the file system shall comply with the client interface defined in Sec. 4.

I6 Data integrity will be assessed by means of the CRC functionality provided in `crc.c`.

I7 The student shall declare auxiliary functions in `auxiliary.h`.

I8 Both the client interface of the file system and the auxiliary functions defined by the student shall be implemented in `filesystem.c`.

I9 The student shall declare auxiliary data types, constants and structures in `metadata.h`.

5.4 Documentation Requirements

D1 Each function in the code shall be properly commented, emphasising obscure implementation details or student-defined behaviour. Programs without comments will have their grade penalised.

D2 A minimal report must include:

- **D2.1** A title page with the name of the authors and their student identification numbers.
- **D2.2** An index.
- **D2.3** The detailed design of the file system, including assumptions, data structures, algorithms and optimisations.
- **D2.4** A high-level description of the core functionality.
- **D2.5** The design of the test plan to verify the functionality of an implementation of the design.
- **D2.6** Conclusions, describing the main problems found throughout the assignment, and how they have been solved. Optionally, here you can include personal comments regarding the project.

[2]It is highly recommended to check against *guernika* throughout the whole development of the assignment. Contact your lab teacher if you have issues or inquiries regarding this platform.

[3]Contact your lab teacher in case you find any bugs or issues you consider not working as intended **BEFORE** modifying the code yourself.

D3 *Every* design decision must be properly justified and contextualised with respect to the theoretical concepts of the subject.

D4 Do not include any source code in the report. If present, it will be ignored.

D5 Do not include any screen shots code in the report. If present, they will be ignored.

D6 Each case in the test plan must specify its objective, procedure, input, and expected output. Optionally, you can also include whether your implementation passes the case.

D7 Every page must be numbered, except the cover.

D8 Text must be justified.

D9 The report must not exceed 15 pages, including cover, figures, tables, and references. Any content exceeding this limit will be ignored.

5.5 Submission Requirements

S1 You must submit your work using Aula Global. The deadline will be indicated in the Aula Global assignment, and notified accordingly. Submitting the assignments by mail is not allowed without prior authorization.

S2 Submission must be done separately for the code and the report.

S3 The report must be submitted as a PDF file through Turnitin. Only PDF files will be graded.

S4 The submission must comply with the following naming conventions, where AAAA-AAAAA, BBBBBBBBB, and CCCCCCCCC are the student identification numbers of the students in each group:

S4.1 Report: `dssoo_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCCCC.pdf`

S4.2 Code: `dssoo_p2_AAAAAAAAA_BBBBBBBBB_CCCCCCCC.zip`

S5 The compressed file shall contain the following files:

S5.1 Implementation of the filesystem: `filesystem.c`

S5.2 Implementation of the test plan: `test.c`

S5.3 Full header file folder: `include`

S6 A maximum of three students is allowed per submission.

S7 All the requirements should be checked before submission.

6 Other Considerations

In addition, the following items need to be considered by the student:

- Provided two file systems, both passing all tests, the better design will achieve higher scores.
- The report is a significant part of the grade given the design-oriented nature of this assignment. Do not neglect its quality.
- Avoid duplicated tests that target the same code paths with equivalent input parameters. The test plan is graded according to its coverage, rather than the number of tests.
- A warning-free compilation does not guarantee that the implementation fulfills the functional requirements. Please test and debug your code properly to assess its behaviour.
- Students are expected to submit original work. In case plagiarism is detected between two assignments both groups will fail the continuous evaluation. Additional administrative charges of academic misconduct may be applied.
- You can use the following command to create the compressed file with the source code:

```
&> zip dssoo_p3_AAAAAAAAA_BBBBBBBBB_CCCCCCCCC.zip filesystem.c
test.c include/*
```