UNIVERSITÉ TOULOUSE III PAUL SABATIER Université de Toulouse

Informative license

ScratchOS: a virtual OS

Topic V0.1

Vincent Dugat

January 2022

IMPORTANT NOTES:

This is a version of the project specification which, like any self-respecting specification, is subject to change based on questions and comments. Updates will be made, if necessary. Check that you are notified of posts on the Moodle forum in order to be notified of new versions.

Project terms:
— The project management part and its documents will be treated in the course "Initiation to project management" and managed by the speakers of this course.
— You will have a tutor for the development and organization part of the code.
— The code is to be programmed in C and Java languages according to the methods described in this document.
— The Moodle page is common to the Project Management part and to coding.
- Normally several we go further. The organization of your development team and its effectiveness as such, is an important point of the project.

Modalities of correction:
— You will have a project management note and a coding note. The coding note includes the receipt (validation) of the code.
— Presence at the recipe is mandatory.
— Both grades count for 50% each in the final grade.
— The total of all the coding points is 80. This score is to be shared between all the members of a team (the final score for each is out of 20 points. We will average it with the project management score itself). even on 20 points). The total will be out of 20 points. A single note will be sent to the administration.
— The splitting of the score between the members of a team is fair by default. This principle can be modified by the development tutor or at the request of the team (it is then desirable that there is consensus). A team member's rating is capped at 20 points.

1 Scratch OS, a (very) simplified UNIX like operating system

We will create, from scratch, an operating system with files, users, a connection procedure, a command interpreter. The hard disk of the machine will be simulated by a file which will be called the virtual disk, which will have to be formatted and which will contain the files and the catalog of the disk. The operating system itself will be a C program, resident in memory, managing tables allowing it to manage a file system, which will be written on a simulated hard disk, for the project, by a file.

The system will manage various operations on files, a table of users, (simplified) rights of the latter on files, a connection procedure and a command interpreter.
As is customary in a system, the operating logic will be divided into layers.

1.1 The virtual DD and its formatting

So the first step in developing our system is file system storage. We are going to use a file which will act as the hard disk. This file will be placed in a directory. The disc will be named *d0*.

In order to be able to use this file to manage the file system of the OS, we define a structure containing the name of the virtual disk file of the system and which is stored from the first block of the disk file. We add a table of structures, similar to the inode table of a UNIX system, and of fixed size. Each entry in the inode table corresponds to a file on disk and gives its name, first block, size, and other information. The whole defines the catalog of the disk.

We consider here that a file name is at most *FILENAME MAX SIZE* characters and that we can have at most *MAX FILES* fishit. _

Files will have a maximum size of *MAX FILE SIZE* _ _

The header file provided contains indicative values for these constants. You can do the project with these values but please make sure that if you change their values, your system remains operational.

1.1.1 Formatting the virtual disk

The source file *cmd format.c*, provided on Moodle as an appendix to this topic, allows you to format the disk. To format your system, the executable requires as a parameter the name of the existing directory in which the disk file will be created, and the size *disk size* of the disc. The size is given in bytes. If the disk *d0* already exists in this directory, it is reset. If it does not exist in the directory, a file whose content is set to 0 is created at the correct size. The system formatting operation must be done only once when creating the system. Any formatting of an existing system will erase its data.

Example : *Syntax*: command name directory file size (bytes)
. /cmd format dir 500000 create file *d0* in directory *direction*, with a size of 500000 bytes.

2 Work to be done in C language

The work to be done is divided and organized into layers as is customary for an operating system. Each layer implements functionality using those of the lower layer and providing services to the upper layer. A file *os defines.h*, provided on Moodle, gives some data structures to use.

Throughout the subject the functions for reading/writing on the virtual disk begin with: write..., the functions for displaying on the screen begin with print...

2.1 Low Level Layer 1: Blocks and Utility Functions

This part aims to write functions to manage the blocks on the system: write, read, erase. The functions suggested here are the minimum useful. You may need other functions.

1. We consider that our system is represented by the global variable *virtual disk sos;*. Before being able to use it, it is necessary to initialize it from the name of the directory containing the formatted virtual disk.

   Write the function *init_disk_sos* which, from the directory name, initializes this variable. At first, we will not initialize the inode table (layer 2). When our system is “off”, it will be necessary to ensure that there is no risk of data loss. To do this, write a function that properly “shuts down” our system, ie saves the catalog on the virtual disk.

2. Write the function *compute nblock* which calculates the number of blocks needed to store a number *not* of bytes.

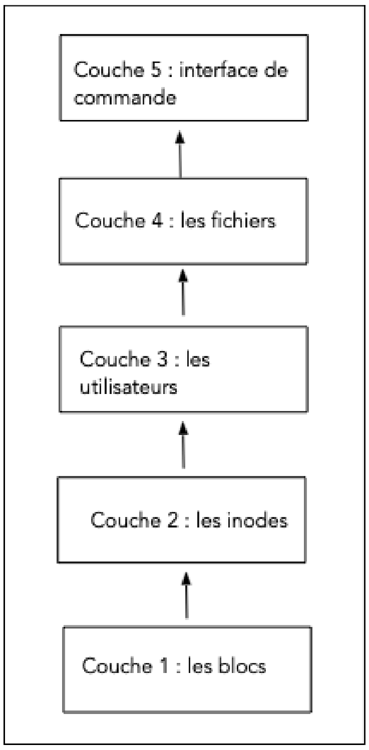

figure1 – Layered organization

3. Write the function *write_block* who writes a block *block*, at position *pos* on the system disk.
4. Write the function *read_block* who reads a block *block* of data, at the position *pos* on the system disk. In the event of a read failure, this function must return an error code that you will specify.

   Remark :C functions *fseek*, *fwrite* and *read* seem best suited for this job.

5. Recommended: Write on-screen display functions of type blocks *t-block* in hexadecimal to verify.

2.2 Layer 2: catalog, super block and inode table management

The file system catalog is a structure containing the information needed to manage files. At this level, some of this information will be completed later.

```
typedef struct inode_s{
    // type file vs dir
    char filename[FILENAME_MAX_SIZE]; // including '\0'
    uint size; // of the file in bytes
    uint uid; //user id owner
    uint uright; //owner's rights between 0 and 3 coding rw in binary uint
    oright; // other's right same
    char ctimestamp[TIMESTAMP_SIZE]; // creation date: 26 bytes char
    mtimestamp[TIMESTAMP_SIZE]; // date of last mod. : 26 bytes uint nblock; // file
    nblock = (size+BLOCK_SIZE-1)/BLOCK_SIZE uint first_byte; // start number of the
    first byte on the virtual disk } inode_t;
```

The low-level operations programmed in the previous part allow to store binary data on the system. In order to be able to structure this data in the form of files, it is necessary to set up an appropriate management system. For this, an inode table[1] fixed size *MAXFILES*, making the association between a filename and its properties, as shown in the code snippet above, is stored at position *INODES START* on the system. An entry of this table contains either a file name and data (non-zero) characterizing its size, the position and the number of blocks used by the file on the system, or a position equal to 0 if it does not designate a file existing in the system.

In order to facilitate access to information, a super block is used which gives the number of system files, the number of users, the number of blocks used and the number of the first free byte in the system. Super block size is fixed (4 blocks on the system)
The objective of this part is to add functions to manage this inode table and the super block.

1. Write the function *write super block* which writes the super block at the very beginning of the disk file.
2. Write the function *read super block* which reads the superblock at the very beginning of the disk.
3. Write a function that updates the field *first free byte* super block.
4. Write the function *read_inodes_table* to load the inode table from the system and use it to complete the initialization for question 1.
5. Write the function *write inode table* allowing the inode table to be written to the system following the super block.
6. Write the function *delete_inode* which, given an index in the inode table, deletes the corresponding inode and compacts the table so that, if *not* files are stored on the system, the *not* first entries in the inode table correspond to these files. The super block will need to be updated.
7. Write the function *get unused inode* which returns the index of the first available inode in the table.
8. Write the function *init inode* which initializes an inode from a file name, its size and its position on the system.
9. Write a program *cmd dump inode* which will be used for the tests on the files. This program takes as argument the name of the directory containing the disk file. After reading the table.

2.3 Layer 3: User management

The system will be multi-user. They will be managed by a user table (an array), the index of each user in the array will serve as *user id* (UID). There is a default user named ”root” which will be created when installing the system, as in a Linux system installation, in case 0 (a #define ROOT UID 0 can be useful).

2.4 Layer 4: File and rights management

Functions from previous layers can now be used to manage files. A file will be described with the structure:

```
typedef struct file_s{
   uint size; // Size of file in bytes
   uchar data [MAX_FILE_SIZE]; // only text files } file_t;
```

We will only manage text files with (very) simplified rights.

1. Write the function *writefile* taking as parameters a file name (character string) and a variable of type *net* containing the file to write to the system. If the filename is not present in the inode table, then a new inode is created for this file and added at the end of the table. The file is written to the system following the files already present. If the filename is present in the inode table, then it is an update and two cases have to be treated:

1. This notion borrowed from UNIX is very simplified here.

— the file has a size less than or equal to the size of the file already present. It is then sufficient to update the data and the inode table (if it is smaller there will be a "hole" on the disk).

— the file is larger than the size of the file already present. You must then delete the corresponding inode then add the file at the end of the disk (the old file leaves a "hole" on the disk).

2. Write the function*read file*taking as parameters a file name (character string) and a variable of type*net*which will contain the read file. If the file is not present on the system, this variable is not modified and the function returns 0. If the file is present on the system, this variable contains the data of the
file read and the function returns 1.
3. Write the function*delete file*taking a file name as a parameter and which deletes the inode corresponding to this file. This function returns 1 if deleted and 0 if the file is not present on the system.
4. Write a function*load file from host*which takes as a parameter the name of a file on the computer you are using (named*host*) and writes it to the system. The name of the file on the system will be the same as on the host.
5. Write a function*store file to host*which takes as a parameter the name of an SOS system file and writes it to the host computer. The file name will also be the same.

To link the files and the users, the inodes integrate the uid of the creator user, his rights (by default rw), and the rights of other users (by default nothing). We can manage these rights by a simple integer between 0 and 3: if we represent each right by a lowercase letter if it is absent and by a capital letter if it is present, we have:
rw = 0 no rights rW = 1
write rights Rw = 2 read
rights
RW = 3 write and read rights

Each inode will also contain the creation date of the file and the date of its last modification (see the function time stamp.c). You can now complete the initialization and management of the inode table.

2.5 Layer 5: the command interpreter

To use all the file system set up we create in this part a basic command interpreter setting up the operating system itself.

Attention :The implementation of the requested commands may require you to go back to the functions already written to add functionality to them.

1. Program a "Unix like" command interpreter according to the loop:
   (a) Reading the command (with its parameters)
   (b) Interpretation
   (c) Execution
   (d) Result Display

   Before entering this loop, the system will request a login and password from the user, calculate the hash of the password, look up the login in the user table, check the hash of the password and if c is compliant, will start the command interpreter. Otherwise, the system re-displays the request for login, password. After three consecutive errors, the system exits.

2. The commands known to the interpreter are: File commands:
   — ls [-l]: list the contents of the catalog. An optional argument for a short (file name, size) or long (all) display.

— cat<file name>:displays the contents of a file on the screen if the user has the rights,
—rm<file name>:deletes a file from the system if the user has the rights,
—cr<file name>:creates a new file on the system, the owner is the user.
—edit<file name>: starsay a file to modify its contents if the user has the rights,
— load<file name>:copies the contents of a file from the "host" system to the system with the same name (similar to a creation),
— store<file name>:copies the contents of a system file to "host" with the same name,
—chown<file name> <login other user>change the owner of a file if the requestor has the rights
—chmod<file name> <straight>change the rights of a file for all other users if the requestor has the rights
—listusers
• quit: exits the command interpreter and the program, saving the file system to disk.

Rights reserved for root:Note: root has all privileges.
— adduser: add a user. The command asks for the login and to create a password.
— rmuser<login>:delete a user

Each command will have to manage the sequence of operations necessary to list the catalog, display, create, delete, etc. A file. After displaying the result or any error message, the prompt will be displayed again. The "quit" command allows you to quit the interpreter and the program after saving the data to disk and closing the files.

Function*hand*will launch this interpreter. The main will take as argument the directory containing the disk file.

3 Installer

Like the installation programs of Linux distributions, it is a question here of creating a program calling some of the preceding functionalities with a*hand*dedicated. This program will be responsible for creating and initializing the file system, creating the root user, asking the user to set a password for this user, creating a file*passwd*containing the root login and the password hash. Save everything on the virtual disk.

Remark :At the end of the installation, the file system contains only one file: the passwd file.

The SOS program will assume the system thus initialized and will read the data directly from it.

3.1 Java program

The Java program consists of implementing tools for analyzing and diagnosing the SOS file system:

1. program a system consistency analysis function by checking:
   — super block information
   — catalog information. For example that the beginning block of an inode added to the size corresponds to the beginning of the next file.
   — user table information.
   This function can call sub-functions.
2. The fragmentation problem: the simplifications that we have introduced for our system generate a strong fragmentation after each file deletion. Write a defragmentation program to compact the storage of files by removing the holes caused by the deletion of a file. The result must be readable by the C program of SOS without alteration or loss of information.

3. Write an interaction graphical interface to manage the various previous commands. Specifications are free.