

Project 0

Setting Up the File System

In this project, you will run through the implementation of a simplified basic file system using a so called “extent-based file allocation policy.” The code and the necessary instructions for compiling it are provided. As you read through the description, you are expected to understand the key processes, as they shall be required for implementing projects 1 and 2 in this course. The implementation is in C++.

Goal

In the design of the file system, we will simulate a disk by a unix file. Please feel free to add/modify parameters and code to the given functions when necessary. In addition, you can define the semantics of the return value by yourself. The description of the provided FileSystem implementation functions are as follows:

1. **Create a file system.** This function will create all the necessary data structures (e.g., for storing free space information), catalog information, etc., which are required for the basic file system. Parameters to the command will include the name of the unix file that has to be supported as a disk, the block size, information about the specific policy for allocating file extents, e.g., best-fit (BF), first-fit (FF), and worst fit (WF). The policy is optional, and its default value is BF. Once the file system has been created, files can be created on the file system.

Example: Create a file system on disk d1:

- (a) Disk size = 128 KB = 2^{17} .
- (b) Block size = 128 bytes = 2^7 .

Command: `createFS d1 17 7`

2. **Add a new disk to the file system.** Once this command is executed, space on the new disk can be allocated to files. Its parameter includes the name of the unix file simulating the disk.

Example: Add new disk d2 to the file system:

- (a) Disk size = 128 KB = 2^{17} .
- (b) Block size = 128 bytes = 2^7 .

Command: `addDisk d2 17 7`

3. **Reorganize the disk.** This command should reorganize the allocated file extents such that the empty (holes) created as a result of say deletion of a file disappear – free space should migrate towards the end of the disk into one contiguous segment. Its only parameter includes the name of the disk to be reorganized.

Example: Reorganize disk d1.

Command: `reorganizeDisk d1`

4. **Create a new file.** Input parameters to this command include name of the file, disk name, size of the initial extent in number of blocks, the size of subsequent extents (that may be requested at a later time), the disk on which the file is to be created (optional), disk on which future extents will be mapped (optional), and whether the file is random access or append only.
Example: Open file `file1` for writing.
Command: `openFile file1 d1 w`
5. **Extend a file.** By executing this command, by default is extended by a secondary extent as specified in the original “create file” command. The size of the secondary extent is also specified in this command.
Example: Extend `file1` with the secondary extent size = 100 blocks.
Command: `extendFile file1 100`
6. **Delete a file.** Its input parameter includes the name of the file.
Example: Delete file `file1`.
Command: `deleteFile file1`
7. **Open a file.** Its input parameters include the file name, and whether the file is opened for reading, writing, or appending. Error checking is done to ensure that overwriting a file that is “append only” is not permitted. The command returns a file descriptor that can later be used by read (“r”), write (“w”), append (“a”) commands to read/modify/append to a file.
Example: Open file `file1` for writing.
Command: `openFile file1 w`
8. **Read a file page from disk.** Its input parameters include a file descriptor returned by the open file command, a block number in the file, and a destination memory address. Usually, a file page is returned to a specified buffer slot. We however assume that the page is directly returned to the user application, (since you will not be implementing a buffer manager at this stage of the assignment.)
Example: Read block number 34 in file `file2` into memory `blk_ptr`.
Command: `readFile file2 34 blk_ptr`
9. **Write a file page back to disk.** Its input parameters include file name, block number, and block of memory containing the new content.
Example: Write the memory block `blk_ptr` into block number 60 in file `file2`.
Command: `writeFile file2 60 blk_ptr`
10. **Append a file.** It writes a block to the end of the file.
Example: Write the memory block `blk_ptr` to the end of `file2`.
Command: `appendPage file2 blk_ptr`
11. **Close a file.** Free the necessary resources.
Example: Close file `file2`.
Command: `closeFile file2`

12. **Destroy a file system.** Free the necessary resources.

Example: Destroy file system `d1`.

Command: `destroy d1`

The commands listed above are a minimal set that basic file systems need to support and are by no means complete. However, even the implementation of the above functionalities should give you a good idea of the complexities involved. For example, we have assumed that the file system is a single user system – adding concurrent processes accessing the files will make the design significantly more complex.

Comments

A very significant part of the design is that of a suitable catalog manager that can store all the necessary information about files, their location on disks, where the extents are stored, etc. This information is referred to as low-level catalog management. Note that the catalog has to be mapped to persistent storage for the file to outlive the process that creates the file. You can assume a special file named `disk0` (system disk) in which you store the catalog.

As the class notes (will) suggest, you can store the information about extents mapped to a particular disk at the beginning of the disk itself.

Finally, at times it is important to recover information about files when the file system itself has gotten corrupted – say the disk containing catalog is failed. In this case, you may still be interested in scavenging the disk for as much information as possible. One way to do this is to associate a page header with each file page and store some information (e.g., the name of the file to which the page belongs, the timestamp when it was created, etc.) in the page header.

Sample Test Command Sequence

Table 1 shows a sample sequence of commands, which you may use to test the performance of the file system. Here are a few comments.

1. When filling blocks for an extent, use the filename append with the block number `I` for each block. i.e. if the file name is "foo" and primary extent size is 4 then the data should look like this:

```
foo1foo1...foo2foo2...foo3foo3...foo4foo4
```

This content allows for easy identification of blocks when reading files in order to prove the correctness of a function.

2. Testify the reorganization is correct. When deleting a file, replace its bytes with empty space characters. In This way, after successful reorganization, the UNIX file that represents the disk will have a straight sequence of numbers (i.e. `foo7 ... foo7` for block number 7) followed by empty spaces. If there are spaces between these numbers, then the reorganization did not succeed. Also re-read all the files to make sure that their data did not change.

