Operating Systems HW #2

Due 22.04.2016, 23:55

In this exercise we develop Vault application that simulates a flat file system. The main purpose of this application is to keep files inside one file. Vault reminds different archive utilities (zip, rar, etc.) but lacks the compression functionality and directory hierarchy. The application manages a *.vlt file performing one requested operation per invocation. A possible file structure is described in Part I, the supported operations are covered in Part III.

Assumptions:

- Stored file names are unique.
- Only regular files are provided and stored.
- Each stored file name length is 256 symbols at most.
- Application is capable to store up to 100 files.
- No file has '<<<<<<<<' and '>>>>>>>>' in its content. We consider these sequences as special delimiter marks.
- Each file can be fragmented internally up to 3 parts at most.
- Once created, vault repository file size doesn't change.
- Persistent time stamps are up to seconds exact.

Part I. Vault repository file structure.

A vault file has the following sections:

Catalog, which consists of

Repository metadata

Vault repository overall file size (ssize_t);
Creation time stamp (time_t);
Last modification time stamp (time_t);
Number of files in the vault (short);

File Allocation Table (*FAT*), where every record is

File name (NULL terminated string, char*);
File size (ssize_t);
File protection (st_mode) - permissions of the original file;
Insertion date stamp (time_t) – when this file was added to this vault;
Data Block 1 offset (off_t) – the absolute offset of the 1st block of the file content in this vault repository;
Data Block 1 size (ssize_t) – the length of the above block;
Data Block 2 offset (off_t) – the absolute offset of the 2nd block of the file content, or zero, if the file isn't fragmented;
Data Block 2 size (ssize_t) – the length of the above, or zero;

...

Data Block 3 offset (off_t) – the absolute offset of the 3rd block of the file content if applicable, or zero, if the file isn't fragmented;
Data Block 3 size (ssize_t) – the length of the above, or zero;

Data blocks
These are just the blocks of data, actual files content, consistent with FAT entries. Each block starts with '<<<<<<<<' and ends with '>>>>>>>>' delimiter marks.

Part II. The flow.

The work with Vault is done by separate invocations. One invocation consists of
initialization – when the Catalog section is read into memory at the beginning, and
performing the operation – either by modifying the vault (i.e. adding/removing/deframenting content, changing FAT), or answering the query (like list, status).

Every performed (even failed) operation outputs the time consumed, in the last line. Copy/paste the relevant code from *timing.c* example in Moodle. Keep the same output format.

Part III. Vault operations.

The following operations are supported

1. Initialization. Allocate a vault repository file. Initialize Catalog section. Fail if the requested file size isn't big enough.
   Example:
   $./vault my_repository.vlt *init* 2M
   ```
   Result: A vault created
   ```

2. Listing files. Printout all FAT entries.
   Example:
   $./vault my_repository.vlt *list*
   ```
   my_file1.txt     3B      0755    Thurs Apr 06 21:18:23 2017
   other_file.txt   15K     0755    Thurs Apr 06 19:24:51 2017
   ```

3. File insertion. Take a file and try to insert it into the vault. Fail if such file_name exists already in the repository. Fail if no free space available. Fail if there is free space but the content has to be fragmented into more than 3 blocks.
   Example:
   $./vault my_repository.vlt *add* /some/path/to/file_name
   ```
   Result: file_name inserted
   ```

4. File deletion. Remove file form the vault. Delete (zero) all the related delimiters in the vault file. Fail if no such file_name in the vault.
   Example:
   $./vault my_repository.vlt *rm* file_name
   Result: file_name deleted

5. File fetch. Create a file with some_file_name in the current directory, with the content and permissions stored in the vault. Fail if no such some_file_name is in the vault. Fail if no write permission for the current directory.
   Example:
   $./vault my_repository.vlt *fetch* some_file_name
   Result: some_file_name created

6. Vault repository defragmentation. Recognize gaps between data blocks. Move data blocks so that no gaps remain. Adjust FAT accordingly.
   Example: $./vault my_repository.vlt *defrag*
   Result: Defragmentation complete

7. Vault repository status retrieve. Print out the number of the files, sum of their sizes, and the Fragmentation Ratio. This number is computed as the following. Find the offset of the first start delimiter ('<<<<<<<<', the offset of the 1st '<'), and the last closing delimiter ('>>>>>>>>', the offset of the last '>'). Compute the distance between the two offsets – consumed length. In the same manner, measure the length of every gap, i.e. the distance between every sequential pair of closing and start delimiters. Sum these distances. Fragmentation Ratio is the ratio of the sum of the gaps and the consumed length.
   Example: $./vault my_repository.vlt *status*
   ```
   Number of files:        2
   Total size:             6023B
   Fragmentation ratio:    0.0
   ```

Remarks and guidelines

- You decide what is the order of the data structures in the vault file.
- You may alternate the described data structures, as long as the assumptions hold.
- You may add additional data structures if you need.
- You decide the algorithms for adding/removing/defragmenting.
- You decide whether a vault file is an ASCII text or a binary file.
- It is possible to store zero-length files in the vault.
- It's OK to have gaps (allocated but unused chunks) in FAT.
- Add "`#define _FILE_OFFSET_BITS 64`" definition somewhere in your code. This helps to work with files greater than 2GB on 32bit Linux. (Check this, for example.)
- You decide how to organize your source. Your code compiles with "`gcc -o vault *.c`" command.
- You submit "ex2_012345678.zip" file. Where "012345678" is your ID (תעודת זהות) number.

We check

- All the operations are supported.
- Proper error handling.
- Correct and efficient memory utilization.
- Minimal and efficient disk access.
- Operation names CaN Be pROVideD iN UPPER, lower or MiXeD caSE.
- The consistency of the data. A fetched file is equal to its original copy.
- The consistency of a vault file in failure scenario. For example, don't leave part of the content if a failure occurred during file insertion, i.e. don't leave starting delimiter without closing delimiter when failing to bring a file into the vault.
- The robustness. If a file is added and removed 10 times nothing breaks or slows.
- Listing and deletion operations are done in O(n) time, where n is the number of files and not their sizes.
- Only data blocks (i.e. actual files content) are between the first '<<<<<<<<' and the last '>>>>>>>>' in the vault file.
- Fragmentation Ratio is zero in the case when several files are added one after another into an empty vault.
- Print the execution time, as mentioned earlier.
- The faster operation performs – the better.