CMPSC 473 – Project #4 – Extended Attributes

Due Date: December 13, 2019 (11:59pm). 60 points

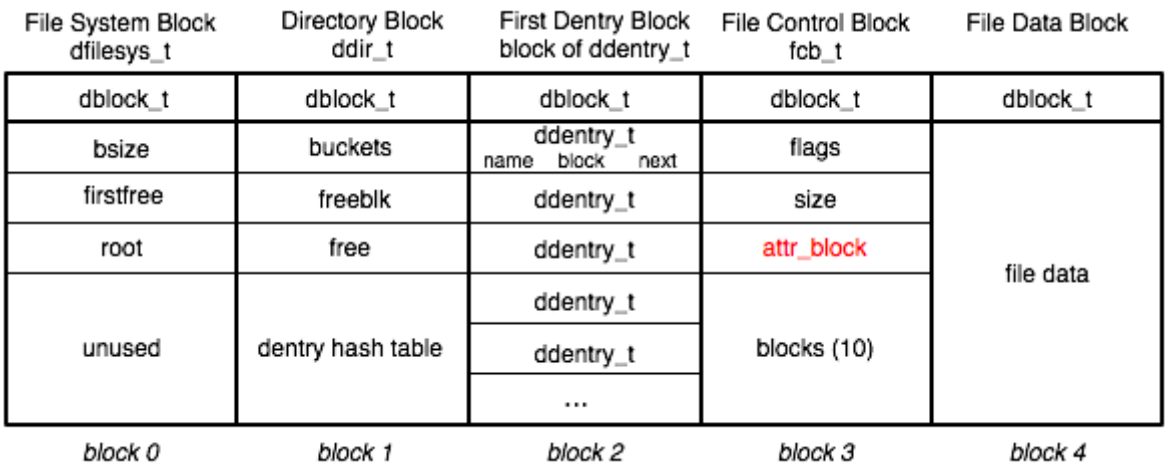
Single person project. **Do your own work!**

In this project, you will extend a provided RAM disk file system with extended attributes that are used to store ad hoc metadata with files. While this code is not based directly on any OS implementation, it does share the common concepts from a UNIX file system.

The File System

The provided RAM disk file system stores its "disk" configuration in memory when you run the project. That is, the layout of the disk is exactly the layout in memory. You will extend this file system with extended attribute that can be added to any file. Fortunately, this extension is largely orthogonal to the provided file system, but there are some lessons to be learned from studying and understanding the file system code provided that you can apply.

The file system is defined by its structure shown below.



The file system consists of a series of **blocks**, which although in memory in your project correspond to the layout of the file system on disk. The main

structures used to implement the different types of file system metadata, including blocks are defined in the file **cmpsc473-filesys.h** in the project code. **Do not modify this file.**

Each block is prefaced by its block metadata specified in a **dblock_t** structure. The block metadata determines the type of block (**free**, which also indicates whether the type is to be determined), the next block in a list of free blocks (**next**), and metadata about the block in a union data structure (**st**), which may either reference a bitmap (for a dentry block) or the end of the block data (e.g., for data blocks).

The last field (**data[0]**) is a reference to the block's data. Note that although this field is declared as an array of size 0, it is actually just a reference to a field whose size is determined at runtime. In the case of blocks, the data in the block is determine by the block type (see BLOCK definitions in cmpsc473-filesys.h). While this may be uncommon in user-space programs, it is common in the Linux kernel – to avoid wasting memory for objects whose size is only known at runtime.

Below, we detail the file system structure shown in the figure above.

- Block 0 is the **file system block** or **superblock**, which defines the overall structure of the file system using a structure of type **dfilesys_t**. This block states the number of blocks in the file system (**bsize** field), the offset in blocks to the next free block (**firstfree** field), and the offset to the root directory block of the file system (**root** field).

- Block 1 stores the **root directory block** of type **ddir_t**, which is the only directory in our filesystem. Each on-disk directory stores a set of references to directory entries (dentries) using a hash table. The on-disk directory stores the number of buckets in its hashtable (**buckets**), the free dentry block (see below) for storing the next directory entry (**freeblk**), and the first free dentry slot index in that dentry block (**free**). The last field (**data[0]** indicates the start of the hash table (the first bucket) for the directory block.

- Block 2 is a **directory entry (dentry) block**, which stores a series of directory entries (dentries) of type **ddentry_t**. The dentries store information about each entry in a directory (i.e., file or subdirectory). In

this project, we only have files, so there are only dentries for files. Each dentry records its file's **file control block** (**block**), which is the first block for a file, the next dentry in the directory's hash table (**next**), and the file name (**name[0]**) and name length (**name_size**).

Thus, directory entry hash table starts in the directory block – by finding the bucket a file name corresponds to – and then traverses directory entries in dentry blocks using the next pointer to find the **next_dentry** block and dentry **next_slot** using the **ddh_t** structures. That is probably the most complicated thing about this file system.

- Block 3 is a **file control block**, which stores the metadata for a particular file using type **fcb_t**. The file control block stores the file permissions available to your one process to this file (**flags**), the size of the file in bytes (**size**), the file data blocks (**blocks**, up to 10 in this project), and the first file attribute block (**attr_block**, see Extended Attributes below). There will be several file control blocks – one for each file created.

- Block 4 is a **data block**, which is used to store file data. The **dblock_t** structure at the start of every block is used to manage the file data, which is written starting in the **data[0]** field, and whose current length is recorded by the **data_end** field (in the union).

Extended Attributes

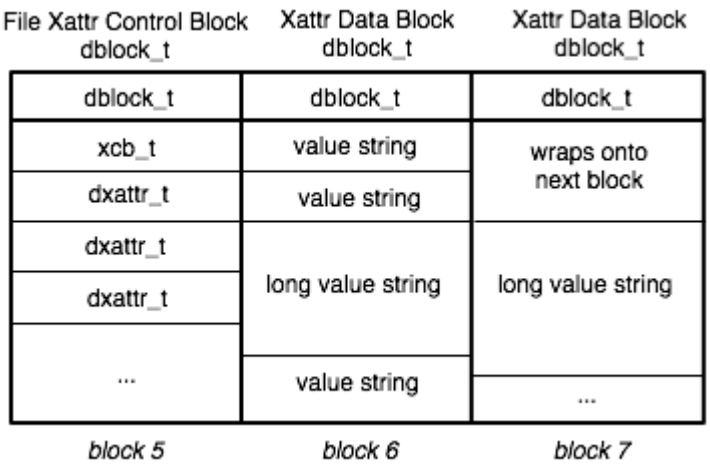
Your task will be to extend the provided file system with extended attributes for files. This section provides background on extended attributes and your specific project tasks.

Background

Many file systems now support extensible storage of file metadata in the form of **extended attributes**. Programs and the kernel may associate attribute–value pairs with a file for any purpose they desire by storing these pairs with the file as extended attributes.

The way the project uses extended attributes largely follows the extended attribute definitions shown in the manpages for fsetxattr and for fgetxattr. The full tarball for the project is available here.

This project will focus on two separate data blocks, **File Xattr Control Blocks (attribute blocks)** and **Xattr Data Blocks (value blocks)**. These are shown in this diagram.



- Block 5 shows a file's extended attribute (xattr) control block. It is a data block (dblock_t) for one file that contains a structure describing the xattr information, called the **xattr control block** of type **xcb_t**. At the end of the xcb_t is a reference to an array of attribute structures of type **dxattr_t**. We will store **all the attribute structures** in this one block. The array of 0 indicates that we do not really know how big these data structures are, nor how many we will really store, so we just leave it to be determined at runtime. This is actually a common practice in the Linux kernel.

- Blocks 6 and 7 are data blocks that store the corresponding attribute values. As values can vary in size, they are just written like a log to the data block. That is, if we first write an attribute **x**'s value **10**, it is written to the data block at the beginning. If we change the value to **11**, then we write this value after 10, and update the location in the attribute structure (value_offset). As shown, values can span multiple value blocks.

- Using the extended attributes for a file entails allocating and initializing an xattr control block (on the first attribute to be set). The xcb_t structure maintains the number of extended attributes stored

(**no_xattrs**) (one dxattr_t entry in this block for each), the size of the dxattrs for this block (**size**), and the references to the value blocks (**value_blocks**) used for the correpsonding attribute values.

For subsequent attributes, create a new dxattr_t structure (after the current ones) and add the value at the end of the xattr data block. Make sure that the dxattr_t for the attribute **name** stores the offset of the **value** in the value blocks (in the field **value_offset**). We can remove a value by setting it to a blank value. We never remove an attribute once created. Consider leveraging **diskWrite** to write the value because the value may span multiple blocks.

In the course of setting attributes, you will have to implement the processing of the flags **XATTR_CREATE** and **XATTR_REPLACE**, where the former requires that no xattr of that name has been created previous (to prevent collisions) and where the latter requires that the xattr already be defined. See more below.

- Getting an attribute value entails, retrieving the file xattr control block, reading the xcb_t structure to find the dxattr_t structure with the **name** string, getting the location of the value string and its size from that dxattr_t structure, and retrieving the corresponding value from the appropriate block. I create the buffer for you and print the value. Consider using **diskRead** as values may span multiple blocks.

Project Tasks

In particular, you are going to be required to implement four functions to enable use of extended attributes.

- **int fileSetAttr(unsigned int fd, char *name, char *value, unsigned int name_size, unsigned int value_size, unsigned int flags)**: This function sets an attribute of **name** (length of **name_size**) of a file specified by the descriptor **fd** to **value** (length of **value_size**) given the **flags** value. The flags values can be **XATTR_CREATE**, which requires that the attribute not be assigned to the file previously, and **XATTR_REPLACE**, which requires that the attribute already has been assigned to the file. Your code needs to return an error if the conditions are not consistent

with the flags. Otherwise, your code should set the attribute's value (more detail below).

A key function of your code will be to retrieve a block to store extended attributes for the file and assign it to the file (file->attr_block). Once assigned, this block should also be made available to the in-memory file (file_t) and the file control block (fcb_t). Then, the attr_block index can then be retrieved from either the file or the disk, but you should look for this value on the file structure before reading from the fcb (the disk). The same should be done for fileGetAttr below.

- **int fileGetAttr(unsigned int fd, char *name, char *value, unsigned int name_size, unsigned int size)**: This function retrieves the value of a file's attribute **name** (length of **name_size**). The function also takes a buffer for the value, called **value**, that is allocated to accept string of up to **size** bytes. Your code should return the number of bytes read into the **value** buffer. If no attribute of **name** is assigned, then nothing (0 bytes) is returned.

- **int diskSetAttr(unsigned int attr_block, char *name, char *value, unsigned int name_size, unsigned int value_size)**: Writes the **value** to a disk data block associating it with the **name** attribute. As described in detail below, **attr_block** is the data block for attribute structures (**dxattr_t**), so **diskSetAttr** must create a structure for **name** if not already there. The attribute values are stored in separate data blocks referenced from the attribute structure.

- **int diskGetAttr(unsigned int attr_block, char *name, char *value, unsigned int name_size, unsigned int size, unsigned int existsp)**: Reads the attribute **name** from the **attr_block** to retrieve the attribute data structure. This structure contains an offset in the value data blocks to enable retrieval of the value which is written to the **value** buffer up to length size. If the **existsp** flag is set, then this function only returns whether the attribute of **name** exists (regardless of whether it has a non-null value).

- Note that reading and writing attributes bears some resemblance to reading and writing generate disk data (although the structures for attributes are different (see below). However, you should use file/diskRead and file/diskWrite for guidance.

In the assignment, an output file (**p4-ooutput** in the tarball) shows the sequence of commands and responses for your file system. You will run 5 commands to generate this output: **./cmpsc473-p4 your_fs cmdi >>& p4-output** where **cmdi** is the ith command for (e.g., cmd1 for the first). This program is deterministic, so your output should match mine (bug disclaimer here).

NOTE: The functionality for cmd1 and cmd2 are part of the provided file system.

Grading:

- Submission builds and executes automatically: 6 points
- fileSetAttr and diskSetAttr expected behavior: 22 points
- fileGetAttr and diskGetAttr expected behavior: 22 points
- XATTR_CREATE and XATTR_REPLACE error cases handled: 10 points

---

Trent Jaeger