Operating systems homework #4: Simple File System Implementation

Introduction

In the previous code work, we made simple shell, and multi-thread programming, which make us recognize that multiple jobs/tasks are sharing physical CPU. Virtual memory homework reveals that physical memory is shared between multiple processes, running on their logical address space.

Based upon that, we begin to look at I/O devices. As an example, we will turn our focus on a storage device and file system, that stores large volume of data. Most operating systems have file system that handles large volume of data in permanent storage. File system provides a concept of 'file', which is the unit of storing information in a computer system.

Basically, data are stored on a storage device that is usually consists of multiple logical blocks. For example, logical HDD blocks are numbered (disk, track, sector) pair. Usually, capacity of permanent storages are larger than the amount of physical memory; thus, only part of storage can be 'loaded' on the memory. That is, when you want to use data in a storage device, you have to load it from the device to memory so that cpu can access to it.

A file system manages the location of stored data in a permanent device, and provides an interface in order for users to easily access the raw data blocks. Imagine that the location of stored data can be various; you cannot memorize all the blocks that you need. In addition, files in computers have variable length; a file can be located across multiple logical disk blocks. Moreover, you can add, remove files, and you can append, edit files.

File system maps data location in a storage with logical location in a file. That is, logical file offset is not always same with physical block numbers, and file system remembers which physical block holds which logical block.

To implement a file system, you may have to understand the concept of meta-data, which is (additional/descriptive) data for data itself (content). For example, let's assume you have file named 'file_1'. Inside 'file_1', you have secret text value, 1234. String 'file_1' is not a data that is directly related with its content '1234'. To read a file that has content '1234', you should know the name of it, the physical blocks 'locations for the file, access permission, and the current offsets, etc. Those additional/descriptive information about the file is called as 'metadata.'

Inside OS, a file is represented with i-node, index node, which holds metadata of the file. Similar to PCB (process control block) that represents a process, i-node represents a file. i-node has rich information about the file, such as file size, access permission, and physical block numbers, etc. When you request to open a file, OS finds the corresponding i-node. Then, your read/write request is handled by invoking reading/writing operation on the storage blocks that are given by the i-node.

I-nodes are also stored in a file system because when you add/remove/edit a file, your file information is also permanently changed. Thus, the device has to store i-node information as

well as storing data itself. Storage defines a special i-node block, which stores only i-nodes. In the i-node blocks, all i-nodes are stored as an array.

In addition to the i-node blocks, storage defines a special block, super block, which stores overall file system related information. For example, super block defines i-node size, number of free i-node blocks, actual use of i-nodes, the first data block number, volume name, etc.

Thus, storage partition (block group) usually looks like the following figure.
[Superblock] [i-node table] [data blocks]
Note that, file system i-node table also has to be loaded in memory from storage.

When OS boots up the system, root file system is mounted. Root file system is the file system that consists the top-namespace of all the files. Usually, the top most file is root directory file, and it is presented with '/' file. When OS mounts the root file system, it populates all files in the file system, and loads i-nodes information from disk so that later files access can be easily made. Superblock usually has root-inode information, i-node number for '/' file.

Directory is a special type of file that has the list of files. A directory is a kind of yellow book that maps human readable filename into index number. Those data are called as 'directory entry.' The directory file includes multiple directory entries, as many as the number of files in the directory.

File can be hierarchically structured, using a directory file. That is, a file is in a directory, and a file can be a directory. If we have a directory file within a directory, it creates 'under' directory, in lower hierarchy.

For example, if you want to open a file '/file1,' you should do the followings.
1. you should mount the root filesystem when you bootstrap your OS. (load i-node table)
2. lookup the / directory file in order to identify the i-node for 'file1.'
    1. look up i-node for '/' directory file.
    2. find out data blocks for '/' file
    3. read in data blocks, and interpret data blocks to directory entry
        1. parse all the data in the block
        2. identify files in '/' directory (dir_entries or directory entries)
        3. match filename in directory entry against 'file1'
    4. find i-node number for file1
        1. directory entry has i-node number
3. mark i-node for '/file1' as opened
    1. additional data is initialized (file offset is set to zero, for supporting data stream)

When you read data on /file1, you should do the followings.
1. you should open the file before you actually use it.
    1. you should have proper i-node pointer
2. identify the blocks to read
    1. check the current read position (file offset)

Submit by July 4th. Firm deadline (no freeday)

2. calculate the starting byte (in a file) to read
3. calculate the starting logical block number (in a file)
4. look up i-node to get the physical block number
    1. i-node has blocks array that maps logical blocks to physical blocks (data block number)
5. read in data blocks
6. copy it to the user buffer
    1. calculate the first byte in data blocks
    2. copy from it

———

In this programming assignment, you will build a simple filesystem. You have to implement above-mentioned simple file system with the proper assumptions.

Extra implementation/analysis is highly encouraged. For example, you can implement user process can make request for read/write for some file, instead of simple I/O. In addition, you can implement files with write operation that requires storage block allocation for additional data, as well as i-node change. You may want to load directory entry into memory because every file access implies addition directory file access; and it will be redundant if you read data from storage at every directory file access, which is one of slowest operations in a computer system. You may want to efficiently search directory entries because there are so many files, you would use has map to easily get the i-node from requested file name. You can link with virtual memory homework. If we read in a disk block, the read disk block can be immediately mapped into a virtual memory with paging system. Metadata block is placed on a memory region called, buffer cache. On the other hand, data block is placed on a memory region called, page cache. Once you have data in the buffer cache or page cache, you don't need to access disk, which is one of the the slowest operations in a computer.

This work requires an amount of time and effort, so don't be too depressed, if you cannot manage this all. Do as much as you can do, and learn from it. (if you need specific guideline, I can give some.)

Lastly, an advice: Begin as early as possible. Ask for help as early as possible. Try as early as possible. They are for your happy-ending.

Submit by July 4th. Firm deadline (no freeday)

Operating systems homework #4: Simple File System Implementation

The followings are specific requirements for your program.

1. The objective: understand file systems structure and organization.
   A. Simulator has to work correctly. (should not break down, obtain exact data that you requested)
   B. Support the following file operations: mount, open, read, close
   C. You should be able to read at least one (normal) file (open, read, close), rather than directory files.
2. Make disk image creation tool
   A. a disk image file contains the raw image of a disk partition: superblock, inode table, data block.
      i. you can use the following definitions, or you can define your own structure
   B. Files population in a disk image.
      i. you can give input file names or directory.
      ii. you can randomly generate filenames, contents.
   C. Your kernel should be able to mount the disk image from your tool.
   D. Output disk image is stored on a file 'disk.img.'

Submit by July 4th. Firm deadline (no freeday)

Operating systems homework #4: Simple File System Implementation

```
/**
  Super block structure
*/
struct super_block {
        unsigned int partition_type;
        unsigned int block_size;
        unsigned int inode_size;
        unsigned int first_inode;

        unsigned int num_inodes;
        unsigned int num_inode_blocks;
        unsigned int num_free_inodes;

        unsigned int num_blocks;
        unsigned int num_free_blocks;
        unsigned int first_data_block;
        char volume_name[24];
        unsigned char padding[960]; //1024-64
};

/**
  32 byte I-node structure
*/
struct inode {
        unsigned int mode;              // reg. file, directory, dev., permissions
        unsigned int locked;    // opened for write
        unsigned int date;
        unsigned int size;
        int indirect_block;     // N.B. -1 for NULL
        unsigned short blocks[0x6];
};

struct blocks {
        unsigned char d[1024];
};

/* physical partition structure */
struct partition {
        struct super_block s;
        struct inode inode_table[224];
        struct blocks data_blocks[4088]; //4096-8
};

/**
  Directory entry structure
*/
struct dentry {
        unsigned int inode;
        unsigned int dir_length;
        unsigned int name_len;
        unsigned int file_type;
        union { // name
                unsigned char name[255];
                unsigned char n_pad[16][16];
        };
};
```

Submit by July 4th. Firm deadline (no freeday)

3. Kernel should mount root file system at system start:
   A. At booting time, the root file system (rootfs) has to be mounted.
   B. Mounting rootfs populates all the files in the root directory, and load i-nodes from disk image. Note that the capacity of a disk is so large that all the contents in a disk cannot be loaded into the memory.
   C. When booting the system, kernel mounts the root file system, and prints the files in the root directory. (similar to ls -al)
4. OS supports file operations: open, read, close.
   A. To access to a file, we need to open before use it.
   B. When a user opens a file, it requires two parameters: pathname and open mode(O_RD), and returns with the file descriptor
      i. Kernel has to read the directory file, and look up i-node number corresponding to the file.
      ii. Check i-node that it is being used by a process.
      iii. Kernel has to allocate a new file descriptor structure, associate it with PCB.
   C. When a user reads a file, it requires three parameters: file descriptor, buffer pointer, read request size.
      i. file structure should have pointer to the corresponding i-node.
      ii. file descriptor structure should have offset value, where the previous file operation has been stopped.
      iii. Kernel has to calculate the logical block numbers to obtain raw data blocks.
      iv. Kernel reads the disk's physical data blocks.
      v. copy it to the user buffer
      vi. return with actual read bytes.
   D. When the read is completed, the user closes the file.
      i. Kernel should mark the i-node that it is available again.
   E. You may assume one user process for this project, and read text data from randomly chosen 10 files.
      i. Your file system would populate files named with file_[n], so that you can easily generate random file names.
5. Extra implementation (directory entry cache/hash)
   A. Directory files are one of the most frequently used file because all the files operations require directory access. Thus, you can cache the directory structure inside the memory, as well as disk.
   B. To quickly access the correct directory entry, we can hash the directory structures. For example, we can make a hash function that take keys from file name, and generate value with the i-node pointer.
   C. Because hash keys are smaller than the actual file names (space), you should check once again for exact file name match.
6. Extra implementation (write operation)
   A. You can simulate implement file write operation. To write to a file, you would need to consider data block allocation if the writing data size is over the data block boundary. In the case, you have to allocate a new data block from a free data block pool, and write it to i-node. Then, you should write data on the block.

Submit by July 4th. Firm deadline (no freeday)

   B. After writing, you should be able to read data after the proper close operation.
7. Extra implementation (work with multiple users)
   A. When there are multiple users, file access should be properly managed by the kernel. Because our previous simulator allows multiple user processes, you can take control with it.
   B. Each pcb should hold open file descriptor structure.
   C. When a user tries to open a file, that is being used by another process, you have two choices.
      i. return failure for open sys call.
      ii. wait the process until the previous user closes the file, and the file is available again.
8. Extra implementation (work with virtual memory)
   A. When you read a disk block, the read block is mapped on virtual memory.
   B. If we use the same disk block, then use the cached memory, instead of reading disk block again.
   C. Prepare 1KB pages.
   D. Read metadata is placed in buffer cache region (virtual memory region inside kernel region)
   E. Read user data is placed in page cache region (virtual memory region in the user region)
9. Extra implementation (making a disk image tool)
   A. populate files from ‘file_1’ to ‘file_100’
   B. each file has three random numbers
   C. generates filesystem image
   D. You can define your own image format; in the case, make some document explicitly explains the structure.
10. Output:
   A. disk image tool (mk_simplefs disk.img …) :
      i. prints out all the files information,
      ii. prints out all the superblock, i-nodes table
   B. You can assume single user process or multiple user processes:
      i. A user process makes open/read/close file operations pair for 10 random files
      ii. A user process prints out each file operations, and file contents on the screen
   C. When you added write operation,
      i. A user process makes open/[read|write]/close file operations for random files
      ii. Kernel prints out all file operations (pid, file op, result with content)
11. Evaluation:
   A. Different credits are given for implementations, and demos.

Happy hacking!

Submit by July 4th. Firm deadline (no freeday)