Operating systems Project #3: Simple File System Implementation

Introduction

In the previous projects, we simulated virtual memory supported multi-process execution environment. Based upon that, we will add functionality that can store a large volume of permanent user data. Most operating systems have file systems to store an amount of data in permanent storage. A file system provides a concept of 'file' to users, which is the unit of storing independent information in a computer system.

Non-volatile (durable even if the power-off the computer) user data are stored on a permanent storage device that consists of multiple blocks. A file system manages the location of stored data in the storage device providing a human-friendly interface to the users.

Different files have different lengths. Also, files can be dynamically created, deleted. Moreover, the file can be edited, and the length of the file could be changed. Thus, the physical location in the storage device is dynamically changed.

Because there are numerous files in a system, and the block locations are dynamically changed, you cannot memorize all the disk blocks. Therefore, a file system in OS remembers the physical location of data in the storage device so that you can find data when you need.

File system maps logical file's locations into the locations in a storage device. That is, the logical file offset is not always the same as the physical device offset, and OS remembers the physical data block that holds the corresponding logical data block. When you read a file, the physical disk block is found based on the mapping information in the file system.

To implement a file system, you may have to understand the concept of meta-data, which is (additional/descriptive) data for data (content). (Not directly related to data content itself) For example, let's assume you have a file named 'file_1'. Inside 'file_1', you have secret text value, 1234. String 'file_1' is not data that is directly related to its content '1234'. To search 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.

Inside OS, files are represented with some number, and the number is used as an index of i-node (index node) data structure, that holds meta-data for a file. (Similarly, PCB represents a process in OS.) i-node has rich information about the file, such as disk data blocks' location, file size, access permission, etc. When you request to open a file, OS looks up the corresponding i-node in the file system. Then, your read/write request is handled by invoking reading/writing operation on the disk block that is given by the i-node.

OS has to store i-node in permanent storage device. That's because when you add/remove/edit a file, your file information is also (permanently) changed. Thus, the device has to update i-node as well as data itself. Storage defines special i-node blocks, which stores only i-nodes. In the i-node blocks, the entire i-nodes are stored as an array.

In addition to the i-node blocks, storage defines a special block, called 'superblock,' which stores overall file system information. Specifically, the superblock defines i-node size, the

number of free i-node blocks, actual usage of i-nodes, the first data block number, the volume name, etc. Thus, storage partition (block group) usually looks like the following figure.
[Superblock] [i-node table blocks] [data blocks]

Note that, the i-node table needs to be loaded into 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, the '/' file. When OS mounts the root file system, OS reads the superblock from the disk, and it populates files in the system by reading i-nodes from disk into memory.

File systems in modern OS support directory structure for structuring file system with a hierarchy. A directory is a special type of file that contains the list of a group of files. In a directory file, all files in the same sub-directory are stored.

In addition, a directory file provides a convenient programming interface to the user. Because OS remembers a file as an i-node number, it is difficult for a human to distinguish files. To aid the human programming usage interface, directory file stores alphabetical name and its i-node number so that a human can easily remember and distinguish different files.

Each element that represents a file in the directory file is called as 'directory entry.' Each directory entry includes (i-node, and file names) for the files in the sub-directory. Although it provides a convenient, human-readable interface, it makes some indirection.

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 to identify the i-node for 'file1.'
    1. I-node of the root directory file '/' is previously determined.
    2. In the '/' i-node, find out data blocks in the disk.
    3. read in disk data blocks into memory, and interpret data to the directory entry
        1. parse all the data in the directory file
        2. identify files in '/' directory
        3. match the filename in directory entry against 'file1.'
    4. find the i-node number for 'file1.'
        1. the 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 use it.
    1. you should have a proper i-node pointer
2. identify the blocks to read
    1. check the current read position (file offset)
    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 of data blocks
    2. copy of it

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

Since this program assignment could be one of your major take-outs from this course, so please work hard to complete the job/semester. If you need help, please ask for the help. (I am here for that specific purpose.) I, of course, welcome any questions on the subject. Note for the one strict rule that you should not copy the code from any others. Deep discussion on the subject is okay (and encouraged), but the same code (or semantics) will result in the sad ending.

Extra implementation/analysis is highly encouraged. For example, you can implement user process that requests of reading for some specific file, instead of doing simple I/O. In addition, you can implement file with write operation that requires storage block allocation for additional data, requiring 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 would be one of slowest operations in a computer system. You may want to efficiently search directory entries because a directory may include many files. So, you would use a hash map to easily get the i-node from the requested file name.

There are various topics related to the file systems, including I/O, scheduling, and virtual memory. Do as much as you can do, and learn from it. (if you need a specific guideline, I can give some.)

Lastly, some advice: begin as early as possible. Ask for help as early as possible. Try as early as possible. They are for your happy ending.

The followings are specific requirements for your program.

1. The objective: understand file systems structure and organization.
    A. The 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. You can assume the following structure or on your own data structure for superblock, i-nodes, and data blocks.
    A. You can use pre-built filesystem image (disk.img), based on the following structure definitions.

due by December 18th

```
/**
  Superblock 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];
	};
};
```

3. Kernel mounts the root file system at system start:
   A. At the 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 the disk image. Note that the capacity of a disk is so large that all the contents of 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. Single user file operations: open, read, close.
   A. Assume a single child process.
   B. The child process conducts only file operations. (open, read, close)
   C. To access a file, we need to open the file before use it.
   D. When a user opens a file, it requires two parameters: pathname and open mode(O_RD), and returns with the file descriptor
      i. The kernel has to read the directory file and look up the i-node number for the corresponding file.
      ii. PCB should have a data structure for the open file. (file structure)
      iii. file structure should have a pointer to the corresponding i-node.
   E. When a user reads a file, the user should provide three parameters: file descriptor, a buffer pointer, read data size.
      i. File structure should have offset value, where the previous file operation has been stopped.
      ii. The kernel has to calculate the logical block numbers to obtain raw data blocks.
      iii. The kernel reads the disk's physical data blocks.
      iv. Copy it to the user buffer
      v. return with actual read bytes.
   F. When the read is completed, the user closes the file.
   G. You may assume one user process for this project, and read text data from randomly chosen ten files.
      i. Look at the directory file for the file names.
5. Extra implementation (directory entry cache/hash)
   A. Directory files are one of the most frequently used files 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 make an in-memory hash function for the directory structure. For example, we can make a hash function that takes keys from the 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 should provide data, along with the file pointer. If you are writing data to a file, the data is written from the beginning of the file.
   B. You may need to consider data block allocation if the writing data size is over due by December 18th

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.

C. After writing, you should be able to read data after the proper close operation.
D. Indirection blocks access to large files can also be considered.

7. Extra implementation (Buffer cache)
   A. To read-in data from disk, OS should prepare free memory region (free page)
   B. You can link the disk operation with virtual memory free page.
   C. When you read-in data, prepare free page frame that can store disk storage block.
   D. The page frame can hold data in the storage, serving as a disk cache.
   E. Then, you can read data from the buffer cache, instead of disk.
   F. Because buffer cache is a cache to a disk, disk content would be different from the buffer cache. You should have a proper synchronization mechanism.
8. Extra implementation (Make disk image creation tool)
   A. A disk image file contains the raw image of a disk partition: superblock, inode table, data block.
      iv. you can use the following definitions, or you can define your own structure
   B. Files population in a disk image.
      v. You can give input file names or directory.
      vi. 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 file ‘disk.img.’
   E. Your file system would populate files named with file_[n] so that you can easily generate random file names.
9. 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 syscall.
      ii. Make the process wait until the previous user closes the file, and the file is available again.
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.

due by December 18th

Operating systems Project #3: Simple File System Implementation

Happy hacking!

due by December 18th