DEPARTMENT OF COMPUTER SCIENCE, HANYANG UNIVERSITY

File System

Operating Systems: Three Easy Pieces

Programming Project with xv6

Final due date: June 23, 2019

1 OVERVIEW

1. File System

   A file system provides an efficient and convenient access to the permanent storage device by allowing data to be stored, located, and retrieved easily. Two main roles of file system are how the file system should look to the user and creating algorithms and data structures to map the logical file system onto the physical-storage secondary devices.

2. File

   Computers can store information on various storage media, such as hard disk drives, solid state drives, etc. So that the computer system will be convenient to use, the operating system provides a uniform logical view of information storage. **The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the *file*.** Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.

   A file is a named collection of related information that is recorded on secondary storage. From a user's perspective, a file is the smallest allotment of **logical** secondary storage; that is, data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs(both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of

bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user. The concept of a file is thus extremely general.

3. Inode

   A file system relies on data structures about the files, beside the file content. The former are called metadata(data that describe data). Each file is associated with an inode, which is identified by an integer number, often referred to as an i-number or inode number. Each inode stores the attributes and disk block location(s) of the file's data.

   The inode number indexes a table of inodes in a known location on the device. From the inode number, the kernel's file system driver can access the inode contents, including the location of the file - thus allowing access to the file.

4. Sync

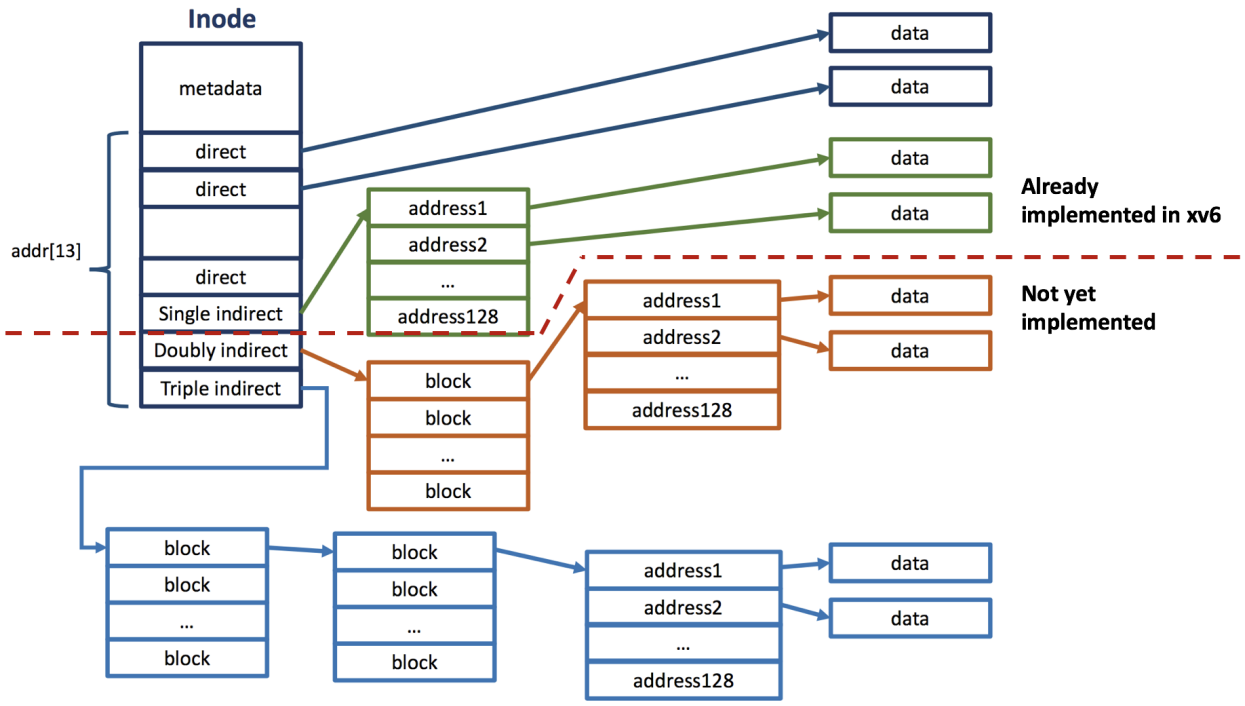
   In Unix operating system, file write operations do not guarantee durability of the written data because of performance problems. Data are written to buffer cache in memory during file write operations, and **sync** operation make all data in buffer cache durable in storage media.

2 IMPLEMENTATION SPECIFICATIONS

1. Milestone 1

   a) **Goal: Expand the maximum size of a file**

      The goal of this milestone is to expand the maximum size a file can have. As you go through the project, you will learn more about how xv6 is building the its file system. The structure of the inode that xv6 currently has is shown below. Refer the figure and Implement a triple indirect block to increase the capacity of the file.


Inode
metadata
direct
direct
direct
Single indirect
Doubly indirect
Triple indirect
addr[13]
address1
address2
...
address128
block
block
...
block
data
data
data
data
Already
implemented in xv6
Not yet
implemented
address1
address2
...
address128
data
data
block
block
...
block
block
block
...
block
address1
address2
...
address128
data
data

b) **Test Program for Milestone 1**

The test program for this milestone is very simple.

Note that you need to build a test program by adding it to the Makefile.

- Create test: write 16 * 1024 * 1024 = 16 Mbytes file
- Read test: read the generated file and verify the data written correctly.
- Stress test: Create and Remove the file 4 times.

  The total writing size is 16 * 1024 * 1024 * 4 = 64 Mbytes.

c) **Tips**

- `struct inode // file.h`
- `struct dinode // fs.h`
- `#define FSSIZE 40000 // param.h`
- `some defined constants // fs.h`
- `static uint bmap(struct inode *ip, uint bn) // fs.c`
- `static void itrunc(struct inode *ip) // fs.c`

2. Milestone 2

a) **Goal: Implement the sync system call**

The goal of this milestone is to implement the sync system call. In original xv6, write system call always try to write data in disk. You should modify the write system call to always write data on buffer cache, and implement the sync system call to flush all data in buffer cache to disk. If log space(or buffer cache) in memory has no more space to save modified blocks, xv6 must clean the log space by flushing the bufferd data.

- int sync(void);
- int get_log_num(void); // return *log.lh.n* value

b) **Tips**

- `struct log // log.c`
- `void begin_op(void) // log.c`
- `void end_op(void) // log.c`
- You don't need to assume multi-thread situation

3 EVALUATION

1. Document

You must write detailed document for your work on the Gitlab wiki. It is evaluated based on that document. If you miss a description for something, it may be not evaluated. Please do careful work that describes your work. The document may include design, implementation, solved problem(evaluating list below) and considerations for evaluation.

2. How to evaluate and self-test

   The evaluation of this project is based on the requirements of the implementation specification above and can be verified through the test-case provided. The points are as follows.

Evaluation items (Milestone 1)	**Points**
Documents	20
Create a large file	40
Read a large file correctly	40
Stress test (Deletion correctness)	20
Total Points	120

Evaluation items (Milestone 2)	**Points**
Documents	20
Write on buffer cache	20
Make data durable by sync call	20
Stress test (Exceed log space)	20
Total Points	80