File System

Operating Systems: Three Easy Pieces

Programming Project with xv6

Final due date: June 15, 2021

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 a 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 the 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 throughout power failures and system reboots.

   A file is a named collection of related information that is recorded on a secondary storage. From a user's perspective, a file is the smallest allotment of a **logical** secondary storage; that is, data cannot be written to a 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 describes the 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 the durability of the written data due to performance issues. Data are written to an in-memory buffer cache during file write operations, and a **sync** operation makes all data in the buffer cache durable in the storage media.

2 IMPLEMENTATION SPECIFICATIONS

1. Milestone 1

   -Due Date: June 8

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

   The goal of this milestone is to implement the sync system call. In the original xv6, write system call always tries to write data straight to the disk. You should modify the write system call to always write data on the buffer cache, and implement the sync system call to flush all data in the buffer cache to the disk. If the in-memory log space(or buffer cache) lacks space to save modified blocks, xv6 must clean the log space by flushing the cached 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 a multi-thread situation

2. Milestone 2

   -Due Date: June 15

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

   The goal of this milestone is to expand the maximum capacity(size) of a file. As you go through the project, you will learn more about how the xv6 builds 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 a file.
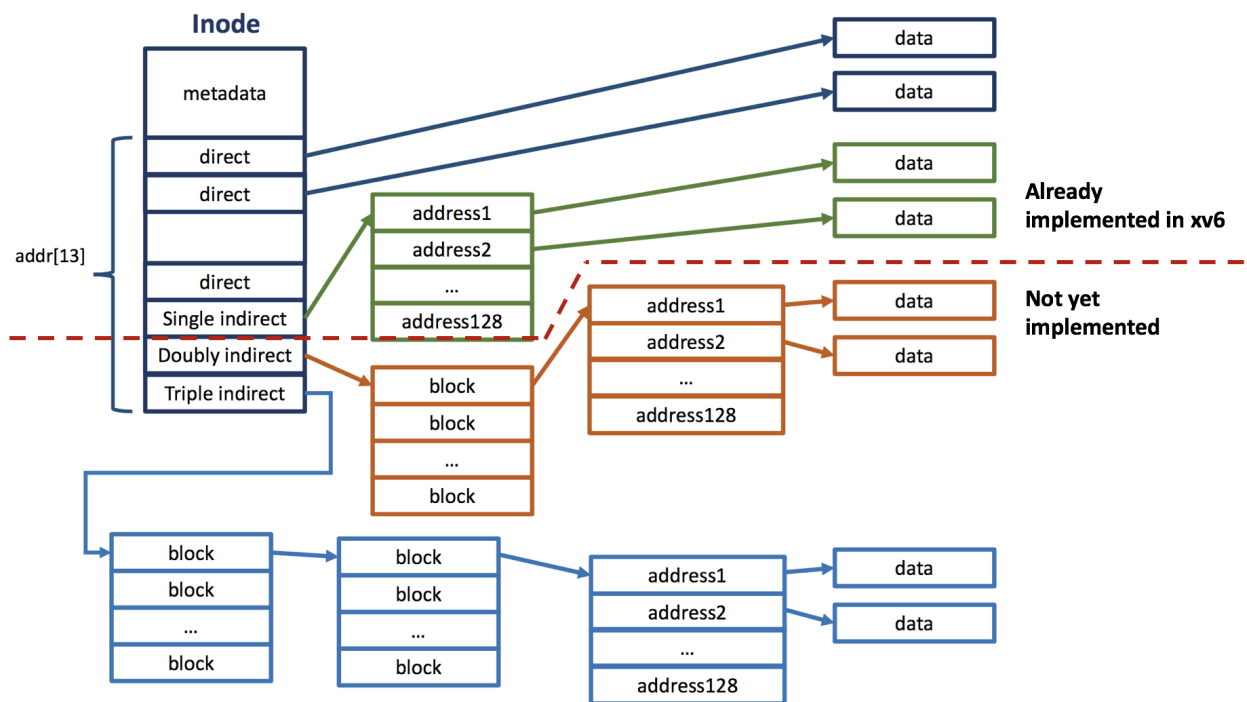
b) **Tips**

- `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`

3. Milestone 3

   -Due Date: June 15

   a) **Goal: Implement the pread, pwrite system call**

   The goal of this milestone is to implement the pread, pwrite system call. While the read, write system call automatically advances the offset of a file, pread and pwrite system call additionally gets the offset parameter to indicate the data offset of the file.

   - int pwrite(int fd, void* addr, int n, **int off**);
   - int pread(int fd, void* addr, int n, **int off**);

   b) **Tips**

   Reference link: https://linux.die.net/man/2/pread

3 EVALUATION

1. Document

   You must write detailed document for your work on the Gitlab wiki. The project is evaluated based on that document. If you miss a description for something, it may be not evaluated. We urge you to do careful work on the documentation. The document may include design, implementation, solved problems(evaluating list below) and considerations for the 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 given as follows.

Evaluation items (Milestone 1)	Points
Documents	20
Write on buffer cache	20
Make data durable by sync call	20
Stress test (Exceed log space)	20
Total Points	80

Evaluation items (Milestone 2)	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 3)	Points
Documents	20
pread works correctly	20
pwrite works correctly	20
Total Points	60