DEPARTMENT OF COMPUTER SCIENCE, HANYANG UNIVERSITY

File System

Operating Systems: Three Easy Pieces

Programming Project with xv6

Final due date: June 21, 2022 (HARD DEADLINE)

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.

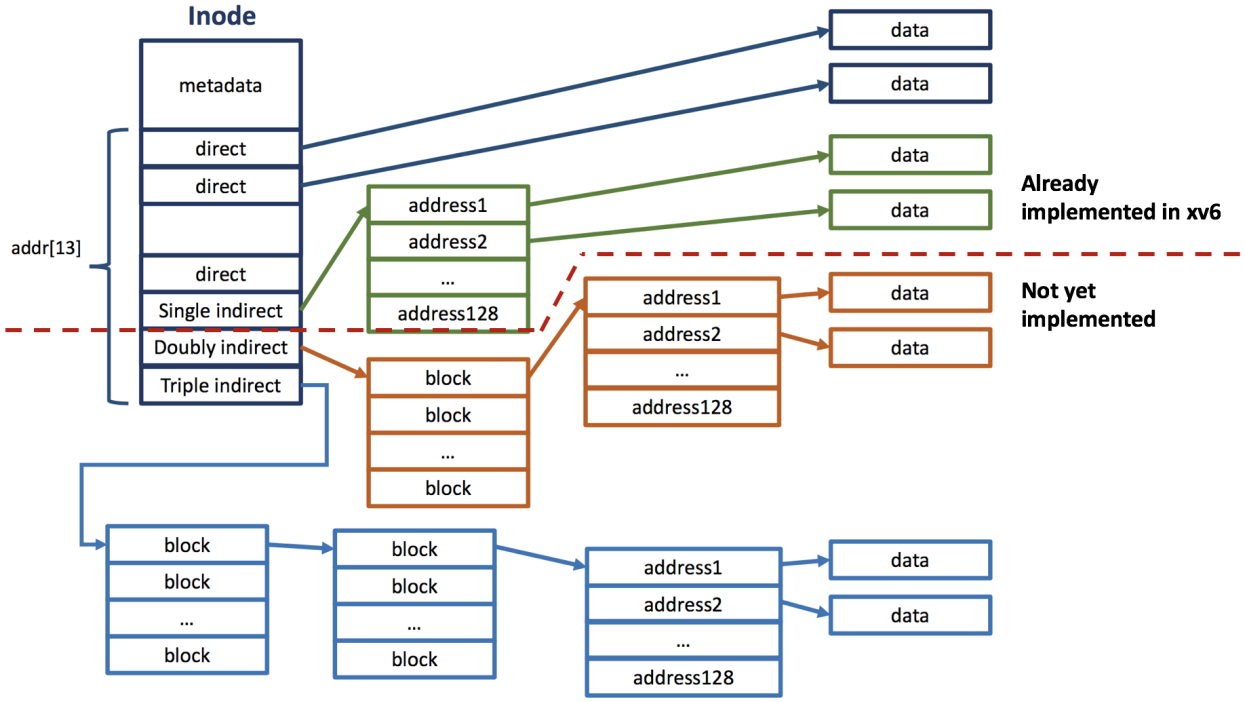
   The inode number indexes a table of inodes in a known location on the device. From the inode number, the kernel's file system driver can access the inode contents, including the location of the file - thus allowing access to the file.

2 Implementation Specifications

1. Milestone 1

   a) **Goal: Expand the maximum size of a file (Due date: June 21, 2022)**

   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 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 maximum capacity of the file.



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 i: Implement the pread, pwrite system call (Due date: June 21, 2022)**

The goal of this milestone is to implement the pread, pwrite system call. While the read, write system call automatically advances the offset of 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

c) **Goal ii: Implement a thread-safe read and write user library**

The goal of this assignment is to implement a thread-safe read and write user library. The pread and pwrite system call implemented in Milestone 2 does not change the offset of the file structure in kernel, but it does not prevent a race condition that occurs when threads in the process write and read the same area of the file. You should implement a user library that can efficiently manage these malicious race conditions.

d) **Build User Library**

The way to implement a user library is explained through the *malloc()* function of xv6. The *malloc()* function, the function that can be used for dynamic memory allocation in the user space, is also implemented in xv6. It cannot be used

for kernel development because it is designed in the form of a library that operates in the user space, but it can be used to implement user programs.
In xv6, user library functions are also declared in *user.h* with kernel system calls. You can see that *malloc()* is also declared in *user.h*. In xv6, most of user library functions are written in the *ulib.c* file. However, it is possible to create a separate file and write the codes of user library functions within this new file. This can also be checked through the implementation of *malloc()* function, which is written in the *umalloc.c* file.
If you want to create new files for user libraries, the newly created files must be included in the build process, so the Makefile must be modified. Specifically, you must modify the contents of the variable that defines the files necessary to build the user library in the Makefile. This variable is ULIB, and the object file must be specified in this variable. Also, the variable to define the file for building user codes must be changed. This variable is the EXTRA variable, which is a function that needs to be changed in the process of building the user program, and the name of the newly created code file should be written in this variable. Examples are as follows:
ULIB = ulib.o usys.o printf.o umalloc.o your_new_file.o
EXTRA = ... cat.c echo.c ... umallo.c your_new_file.o ...

e) **The thread-safe read and write**

The new library should efficiently avoid malicious race conditions, when different threads read and write to the same file within the process. For this, reads and writes to overlapping regions within the process must be protected in the same way as reader-writer locks. That is, different reads of overlapping areas should be performed without waits. However, if a write is performed on the overlapping area, this operation must wait the previously performed reads or writes operations, and subsequent read and write operations on the same area must also wait for this write operation. Inside this library, reading and writing must be performed by calling the system call and multiple threads simultaneously performing I/O without pwrite and pread can modify the file offset of file structure.The use of pread and pwrite system calls is recommended to avoid problems caused by changing file offsets during this process. In other words, milestone2 should proceed with milestone3.

f) **APIs**

The above operations should be implemented in the form of a user library. The following are variables and APIs that should be provided to library users. Note that since it is a user program, it is possible to use functions such as malloc.

i. thread_safe_guard : Thread safety is guaranteed through the thread_safe_guard structure. This structure is allocated by the same descriptor within the process.

typedef struct your_structure thread_safe_guard;

ii. init : The thread_safe_guard_init function allocates a thread_safe_guard structure and maps the allocated structure with the file descriptor received as a parameter. Then, the the address of this structure is returned.

thread_safe_guard * thread_safe_guard_init(int fd);

iii. read : The thread_safe_pread function uses the address of thread_safe_guard returned from the thread_safe_guard_init function. By using this, this function can identify the file to be read. The first variable of this function, file_guard, means the address of the guard allocated by the thread_safe_guard_init function. The second variable, addr, is the address of the memory from which to read the file. The third variable, n, means the number of bytes of data to read. The fourth variable, off, means the location of the data to be read within the file, that is, the offset of the data. And the return value is the size of the read data.

int thread_safe_pread(thread_safe_guard* file_guard, void* addr, int n, int off);

iv. write : The thread_safe_pwrite function uses the address of thread_safe_guard returned from the thread_safe_guard_init function. By using this, this function can identify the file to be read. The first variable of this function, file_guard, means the address of the guard allocated by the thread_safe_guard_init function. The second variable, addr, is the address of the memory where the content to be written is located. The third variable, n, means the number of bytes of data to be written. The fourth variable, off, is where the data will be written on storage.

int thread_safe_pwrite(thread_safe_guard* file_guard, void* addr, int n, int off);

v. destroy : The thread_safe_guard_destroy function deletes the thread_safe_guard structure allocated from the thread_safe_guard_init function. To do this, it receives the address of the thread_safe_guard structure allocated as a parameter, and uses this address to destroy the previously allocated thread_safe_guard structure.

void thread_safe_guard_destroy(thread_safe_guard * file_guard);

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
pread works correctly	20
pwrite works correctly	20
thread-safe pread works correctly	10
thread-safe pwrite works correctly	10
Total Points	80