TDDB68/TDDD47 - Lab 6

Felipe Boeira

February 2021

1 Goal

The goal of this assignment is to learn how to organize concurrent accesses by multiple programs to the file system such that you preserve the data consistency during read/write operations and do not corrupt the file system structure.

2 Overview

This assignment covers:

- Synchronization of files: concurrent access to the files in the file system with `read/write`
- Synchronization of the file system: modification of the existing structure with `remove/create` and access with `open/close`

2.1 User Programs

In the previous assignments (Labs 3-5), you implemented synchronization of several user programs, however, the synchronization of files and the file system was out of scope. In this assignment, you are supposed to implement the readers-writers synchronization algorithm for performing reading and writing operations on several files by several programs. Each file can be open by several programs and even by the same program several times. Since the user programs can concurrently modify not only files but the file system itself, you need to synchronize the corresponding system calls (`remove/create` and `open/close`) to preserve consistency of the file system.

2.2 System Calls to Provide Access to Files and the File System

In this assignment, you will need to implement four new systems calls:

- seek - sets position in a file for read and write system calls.

- tell - returns the position in a file.
- filesize - returns the file size.
- remove - removes a file from the file system.

You will also need to provide synchronization to the following system calls implemented in the previous labs:

- read - reads from a file or the console (the keyboard).
- write - writes to a file or the console (the monitor).
- open - opens a file.
- close - closes an open file.
- create - creates a new file.

All these system calls will allow to perform the majority of file operations. Note that the file size is still considered to be fixed in this lab for the sake of simplicity.

3 Preparations

Pintos File System is a Unix-like file system, which is close to one described with the Virtual File System (VFS) interface. So, read Chapter 21.7.1 "The Virtual File System" of the course book. Pintos uses the same concept of inodes, open files, superblocks and dentry objects. The two last ones correspond to disk and directory in Pintos.

The synchronization of concurrent access to files (reading/writing) is one of the basic issues in operating system design. Usually, it is called "readers-writers problem". The problem with the possible solutions is described in Chapter 6.6.2 "The Readers-Writers Problem" of the course book.

The directory is a special file, which contains file names and file locations on the disk. In other words, it associates the file name with the actual file placement on the disk. Since the kernel and multiple user programs can access the directory concurrently (while creating, removing and opening files) it also needs to be synchronized. Moreover, the operating system keeps track of currently free disk sectors. Creation and removal of files change the map of free disk sectors, which also requires synchronization.

4 Preparatory Questions

Before you begin doing your lab assignment, you have to answer the following set of questions to ensure that you are ready to continue:

- One may synchronize access to files by locking the whole file while reading/writing. Think about why it is not a good idea.
- What is the readers/writers problem? Which modifications of readers-writers synchronization algorithm exist? Find pseudo code for the algorithm that prioritizes readers.
- Think of a scenario where the concurrent access to a file system with no synchronization causes a problem such as inconsistency or corruption of the file system.
- What is the difference between the inode and the file object?
- Consider the following set of actions, which are provided in the following chronological order. What will student.txt contain?

Process A: create(student.txt, 1000)
Process A: fd = open(student.txt)
Process B: remove(student.txt)
Process C: create(student.txt, 1000)
Process C: fd = open(student.txt)
Process C: write(fd, "AAA", 3)
Process A: write(fd, "BBB", 3)
Process A: close(fd)
Process C: close(fd)

4.1 Source Code

You will need to use the functions and the structures provided in `filesys/file.[h|c]` and `filesys/filesys.[h|c]`. So, clear understanding of what those functions are doing is essential for completing this lab assignment. You will also need to have a look into `filesys/directory.[h|c]` in order to get some ideas how the directory is implemented.

Implementation of the Pintos File System is already done in the following set of files:
`filesys/file.[h|c]` - operations on files. A file object represents an open file. Read the description and understand the major steps at least of the following functions: file_open, file_read, file_read_at, file_write, file_write_at, file_length, file_seek, file_tell, and file_close.
`filesys/filesys.[h|c]` - operations on the file system. Read the description and understand the major steps at least of the following functions: filesys_open, filesys_create, and filesys_remove.
`filesys/directory.[h|c]` - operations on directories. It is required to have some understanding of the functions dir_open_root, dir_lookup, dir_close, dir_remove and dir_add in directory.c that are called from filesys_open, filesys_create

and filesys_remove. You should have a clear picture of how the file entry is added to and removed from the directory.
`filesys/inode.[h|c]` - the most important part of the implementation related to the file system. An inode object represents an individual file. Understand when and why the open_cnt counter (property of inode structure) is increased and decreased in inode_open and inode_close. When we want to delete the inode, it is first marked as "to be deleted" with inode_remove and then it is deleted in inode_close when open_cnt becomes 0. The inode functions are called by the wrapper functions implemented in filesys/directory.[h—c], filesys/-file.[h—c], and filesys/filesys.[h—c].
`devices/disk.[h|c]` - implementation of the low-level access to the disk-drive. You should not use these functions directly in your code.
`filesys/free-map.[h|c]` - implementation of the map of free disk sectors. Read the specification of free_map_allocate and free_map_release (reading the implementation of these functions is not required).

Before you proceed to the implementation part of this lab assignment, answer on the following control questions:

- What is the difference between file_open and filesys_open?
- Which functions from inode.c are called when you call filesys_remove, filesys_open, file_close, file_read, and file_write?
- When you remove the file, what is removed first, the file name from the directory or the file content from the disk? How and when is the file content removed?
- What happens if you attempt to remove an open file?
- How can you keep track of the position in a file?
- Can you open a file, on which filesys_remove has been called?
- Find where free_map_allocate and free_map_release are used in inode.c.
- There are few levels where you can add your implementation of the readers-writers problem: system calls, files, and inodes. Think about advantages and disadvantages of each approach. Which level is the most appropriate? Motivate your answer.
- Find the places in the code, where the disk is accessed outside `read/write/open/close/create/remove` system calls. Reconsider your motivation for the previous question.

5 Assignment in Detail

The main part of this assignment is to implement (or extend) the following system calls:

`int read (int fd, void *buffer, unsigned size)`
Reads size bytes from the file with identifier fd into buffer. Returns the number of bytes actually read (0 at end of file), or -1 if the file could not be read (due to a condition other than end of file). Fd 0 reads from the keyboard. Several readers should be able to read from a file at the same time. However, reading should be forbidden if the file content is being changed by the writer.

`int write (int fd, const void *buffer, unsigned size)`
Writes size bytes from buffer to the open file fd. Returns the number of bytes actually written or -1 if the file could not be written. Writing past end-of-file would normally extend the file, but the file growth will not be implemented. When fd=1 then the system call should write to the console. Only one writer can write to a file at the same time. The writer must not write if at least one reader is reading from the file.

`int open (const char *file)`
Opens the file called file. Returns a nonnegative integer handle called a "file descriptor" (fd), or -1 if the file could not be opened.

Within each process, every call to open returns a unique ID (even for the same file) and associates a distinct position for reading/writing.

It should not be possible to open the file, on which remove has been called but the actual deletion has not been done yet (for more details, look into the description of remove system call). This part of functionality is already implemented in Pintos (look into `filesys/inode.[h|c]`).

`void close (int fd)`
Closes file descriptor fd.

`void seek (int fd, unsigned position)`
Sets the current position in the open file fd to position. If the position exceeds the file size, it should be set to the end of file.

`unsigned tell (int fd)`
Returns the current position in the open file fd.

`int filesize (int fd)`
Returns the file size of the open file fd.

`bool create (const char *file, unsigned initial_size)`
Creates a new file called file initially initial_size bytes in size. Returns true if successful, false otherwise.

```
bool remove (const char *file_name)
```

Removes the file with the name `file_name`. Returns true if successful, false otherwise.

Note that the open files must not be deleted from the file system before they are closed. All the processes, which have this file opened when remove is called, can work with the file as usual until they close it. The operating system should wait until the file is closed by all processes, which have already opened it, and only then perform the actual deletion of the file content. In case the file has to be deleted but the actual deletion is postponed, no process can open this file. This part of functionality is already implemented in Pintos (look into `filesys/inode.[h|c]`).

Hint: Make sure that the relevant synchronization primitives for the readers-writers problem will be shared among all current and coming open instances of the particular file.

6 Test Programs

The following tests should pass if your implementation is correct in addition to the tests from previous labs:

tests/filesys/base/lg-create
tests/filesys/base/lg-full
tests/filesys/base/lg-random
tests/filesys/base/lg-seq-block
tests/filesys/base/lg-seq-random
tests/filesys/base/sm-create
tests/filesys/base/sm-full
tests/filesys/base/sm-random
tests/filesys/base/sm-seq-block
tests/filesys/base/sm-seq-random
tests/filesys/base/syn-read
tests/filesys/base/syn-remove
tests/filesys/base/syn-write
tests/userprog/close-twice
tests/userprog/read-normal
tests/userprog/multi-recurse
tests/userprog/multi-child-fd

In order to run the tests you need to do the following:

Copy this **Make.tests** to src/tests/userprog.
Copy this **Make.vars** to `src/userprog`.
Go to `src/userprog`.

Clean up everything with `make clean`.
Run `make`.
Go to `src/userprog/build`.
Create a new simulated disk by following the instructions from Lab 1.
Run `make check`.
All 66 tests should pass if the implementation is correct.

For testing your readers-writers algorithm, we provide the following user programs: `pfs`, `pfs_reader`, `pfs_writer`. These programs run several readers and writers accessing the same file. In order to run these programs, you should run the `start_pfs.sh` script in src/userprog.

Our ultimate concurrency test is the *recursor ng* test. This recursively creates children and waits for them, check the source code in src/examples for more details. To run it, execute the `start_recursor.sh` script in src/userprog.

In case you are using the Virtual Machine, make sure you pull the latest changes from the original repo or add the '-v' flag to the pintos commands in the test scripts. If any of the pfs or recursor ng tests fail, you will get errors printed to the console.

7 Helpful Information

Code directory: `src/userprog, src/filesys, src/devices, src/threads, src/lib, src/lib/kernel`
Textbook chapters: Chapter 10: File System
Chapter 11: Implementing File-Systems
Chapter 6.2: The Critical-Section Problem
Chapter 6.7: Classic Problems of Synchronization
Chapter 16.7.1 The Virtual File System
Documentation: Pintos documentation related to Project 2

(Always remember that the TDDB68 lab instructions always have higher precedence)

8 Acknowledgement

This document is based on previous content from the TDDB68 website.