Project 4: File System

author: 冯宇凡 fengyf2@shanghaitech.edu.cn

杨　斌 yangbin@shanghaitech.edu.cn

date: 2022/01/01

Consulted Sources:

- CSCI 350: Pintos Guide

  https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Guide

- Other guy's implementation

  https://github.com/guliashvili/CS140-Operating-Systems-pintos

  https://github.com/wookayin/pintos

INDEXED AND EXTENSIBLE FILES

DATA STRUCTURES

> 1. Copy here the declaration of each new or changed `struct` or `struct` member, global or static variable, `typedef`, or enumeration.
>
>   Identify the purpose of each in 25 words or less.

In `filesys/inode.h`:

```
/* Blocks stored in one indirect block. */
#define BLOCK_IN_INDIRECT 128

/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
    block_sector_t dindirect_block;     /* double indirect block sector. */
    off_t length;                       /* File size in bytes. */
    off_t capacity;                     /* inode capacity */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[124];               /* Not used. */
  };
```

> 2. What is the maximum size of a file supported by your inode structure? Show your work.

Our `inode_disk` structure contains one doubly indirect block.

The size of one block is 512 bytes and size of a `block_sector_t` variable is 4 bytes. For a doubly indirect block or an indirect block, it can store 128 `block_sector_t` variables. So, our doubly indirect block can reference to $128 = 2^7$ indirect blocks, and they can reference to at most $128 \times 128 = 2^{14}$ blocks. That is memory space of $2^{14} \times 2^9 \text{ bytes} = 2^{23} \text{ bytes} = 8 \text{ MB}$. Therefore, the maximum size supported is 8MB.

SYNCHRONIZATION

> 3. Explain how your code avoids a race if two processes attempt to extend a file at the same time.

File extend will only be called in `inode_write_at()` function. This means that file will only be extended when a process is trying to write to this file. However, we have a lock for file system in `syscall.h` to ensure only one process can write to file. Also, inode itself has `inode->deny_write_cnt` to ensure there can't be processes write to a file at the same time.

> 4. Suppose processes A and B both have file F open, both positioned at end-of-file. If A reads and B writes F at the same time, A may read all, part, or none of what B writes. However, A may not read data other than what B writes, e.g. if B writes nonzero data, A is not allowed to see all zeros. Explain how your code avoids this race.

As mentioned above, we have a lock for file system in `syscall.h` to ensure that only one process can access to the file at the same time. Therefore, our code will not encounter this kind of situation that A reads and B writes.

> 5. Explain how your synchronization design provides "fairness". File access is "fair" if readers cannot indefinitely block writers or vice versa. That is, many processes reading from a file cannot prevent forever another process from writing the file, and many processes writing to a file cannot prevent another process forever from reading the file.

We keep the filesys lock holder list with FIFO policy, so a thread will not be waiting forever.

RATIONALE

> 6. Is your inode structure a multilevel index? If so, why did you choose this particular combination of direct, indirect, and doubly indirect blocks? If not, why did you choose an alternative inode structure, and what advantages and disadvantages does your structure have, compared to a multilevel index?

Yes, we use a doubly indirect blocks with enough blocks to hold an 8M file.

SUBDIRECTORIES

DATA STRUCTURES

> 1. Copy here the declaration of each new or changed `struct` or `struct` member, global or static variable, `typedef`, or enumeration.
>
>   Identify the purpose of each in 25 words or less.

In `filesys/directory.h`:

```c
/* Type of directory entry. */
enum entry_type
{
  DIRECTORY,
  FILE
};

/* A single directory entry. */
struct dir_entry
{
   block_sector_t inode_sector;        /* Sector number of header. */
   char name[NAME_MAX + 1];            /* Null terminated file name. */
   bool in_use;                        /* In use or free? */
   struct dir *parent;                 /* The parent directory of the entry */
   enum entry_type type;               /* The type of this entry, file or directory */
};
```

In `threads/thread.h`:

```
struct thread
{
    struct dir* current_dir;            /* Current directory. */
}
```

ALGORITHMS

> 2. Describe your code for traversing a user-specified path. How do traversals of absolute and relative paths differ?

This is implemented in `filesys/directory.c:dir_lookup()`. We basically tokenize the path by '/' and look up if this token (name) exists in the current parent directory. If it exists and it is a directory, then set current parent directory to this directory and look for the next.

The only difference is that we will check if the first character is '/'. If it is, then our current parent directory should be root directory. Otherwise, it will be our input current directory.

SYNCHRONIZATION

> 3. How do you prevent races on directory entries? For example, only one of two simultaneous attempts to remove a single file should succeed, as should only one of two simultaneous attempts to create a file with the same name, and so on.

First, we keep a filesys lock in `userprog/syscall.c, so there will be only one thread doing the creating or removing. Then before changing the filesys, we will look up the filesys to prevent removing or creating a same file twice.

> 4. Does your implementation allow a directory to be removed if it is open by a process or if it is in use as a process's current working directory? If so, what happens to that process's future file system operations? If not, how do you prevent it?

No. Before removing the directory, we will check its open count and check whether it is the current working directory. If this directory can't be removed, we will return false.

RATIONALE

> 5. Explain why you chose to represent the current directory of a process the way you did.

We add a pointer to the current directory in the thread's structures. Because in this way, the directory will be easy to find and lookup in it.

BUFFER CACHE

DATA STRUCTURES

> 1. Copy here the declaration of each new or changed `struct` or `struct` member, global or static variable, `typedef`, or enumeration.
>
>   Identify the purpose of each in 25 words or less.

In `filesys/cache.h`:

```
/* Cache entry number in cache table. */
#define CACHE_SIZE 64

/* Cache table entry. */
struct cache_table_entry
```

```
{
    block_sector_t sector;              /* The sector of the data store */
    uint8_t data[BLOCK_SECTOR_SIZE];    /* The data in memory */
    bool dirty;                         /* Dirty flag */
    bool valid;                         /* Valid falg */
    bool last_used;                     /* Last used flag */
};

/* Cache table of CACHE_SIZE entries. */
struct cache_table_entry cache_table[CACHE_SIZE];

/* Lock for cache table. */
struct lock cache_table_lock;

/* Next id to try evicting in cache table. */
size_t cache_table_evict_id;

/* Used length of cache table */
size_t cache_table_used;
```

ALGORITHMS

> 2. Describe how your cache replacement algorithm chooses a cache block to evict.

We use second chance algorithm to choose a cache block. We newly define a global pointer `cache_table_evict_id`.

Every time we want to choose a cache block to evict, we continue to move the pointer (`cache_table_evict_id++`) along the cache table. When we encounter a cache table entry with `valid == true`, we set `last_used` to 0 if `last_used == 1` and choose the entry to evict if `last_used ==  0`.

> 3. Describe your implementation of write-behind.

We set a dirty bit in our entry. When `cache_block_write` is called, we only write to cache and set dirty bit to dirty. And when this entry is being evicted, or when we close the inode or file system, it will be write back to disk.

> 4. Describe your implementation of read-ahead.

We create a thread to do read-ahead. That when reading, we will automatically use this thread to read the next block of file into cache.

SYNCHRONIZATION

> 5. When one process is actively reading or writing data in a buffer cache block, how are other processes prevented from evicting that block?

We use a lock for cache table to ensure that only one thread could use `cache_block_write` or `cache_block_read`. So when one process is reading or writing data, no cache entry will be evicted.

> 6. During the eviction of a block from the cache, how are other processes prevented from attempting to access the block?

As memtioned above, we execute the evict in read and write and use a lock to ensure only one thread could read or write at the same time.

RATIONALE

> 7. Describe a file workload likely to benefit from buffer caching, and workloads likely to benefit from read-ahead and write-behind.

From buffer caching: when we need to repeatedly access some files that smaller than cache.

From read-ahead: when we need to read a large file block by block.

From write-behind: when we need to write to the same file again and again.

Contribution

- Feng Yufan: inode and buffer cache.
- Yang Bin: Subdirectory.
- Debug together.