CS130 Project 4 Design Document

Group16

- Cunhan You: youch@shanghaitech.edu.cn
- Junda Shen: shenjd@shanghaitech.edu.cn

Reference:

> Pintos Guide by Stephen Tsung-Han Sher

Task 1: Indexed and Extensible Files

Data Structure

A1: 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.

```
struct inode_disk
{
  block_sector_t direct_blocks[DIRECT_BLOCKS_COUNT];
  block_sector_t indirect_block;
  block_sector_t doubly_indirect_block;

  bool is_dir;                        /* Is dir? */
  off_t length;                       /* File size in bytes. */
  unsigned magic;                     /* Magic number. */
};

/* Sturcture containing indirect blocks */
struct inode_indirect_block_sector
{
  block_sector_t blocks[INDIRECT_BLOCKS_PER_SECTOR];
};
```

A2: What is the maximum size of a file supported by your inode structure? Show your work.

The inode block of a file consists of 123 direct blocks, one indirect block and one double indirect block. An indirect block consists of 128 blocks, which can contain 128 sections in total, and a double indirect block can contain $128^2 = 16384$ sections.

Therefore, the maximum size of a file is $123 + 128 + 16384 = 16635$ sectors, i.e. 8.12255859375 MB.

Synchronization

A3: Explain how your code avoids a race if two processes attempt to extend a file at the same time.

The `file_lock` at the system call level prevents two processes writing or reading files at the same time, so it is impossible for them to extend a file at the same time.

A4: 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.

Even if different processes access files at the same time, data race will not occur. All the system calls related to the file system (including the system calls related to the newly added directory) are atomic by the `file_lock`.

A5: 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.

The `file_lock` at the system call level is acquired every time a process want to do file operations(read/write), and released the lock immediately after operation.

Rationale

A6: 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, it is a 3-level index inode structure.

According to the requirements, the file size must support more than 8 MB, and it needs to support more than $8 \times 2^{20} / 512 = 16384$ blocks. The size of inode block is 512 bytes, so at least double indirect block should be used.

Task 2: Subdirectories

Data Structures

B1: 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.

```
struct thread
  {
    /* Current working directory --for proj4 */
    struct dir *cwd;
  };

struct file_descriptor
  {
    /* dir for a file, null if it is just a file not a dir */
    struct dir* dir;
  };
```

Algorithms

B2: Describe your code for traversing a user-specified path. How do traversals of absolute and relative paths differ?

The function is called `dir_open_path` in `directory.c`.

It open the directory indicated by the specified path string and return its `dir` pointer. The implementation of this function is to use `strtok_r` to divide the '/' into separators and follow the directory structure through `dir_lookup()`. In the small part, there is a deleted directory to open, which is also handled. The deleted directory makes open fail.

We require a path, an string to store the parsed dir returned by `name_resolution`.

We first copy the path for tokenizing.

Then, check if it's absolute path, if so, we open the root.

Otherwise we open the working direct of the process.

Then we parse the filepath from the beginning, each time a string before the next '/', check if there is the required named file or dir in the current dir and the parsed filename is a dir name.

The difference between traversals of absolute and relative paths is whether the parse start from the root or the working dir of the caller process.

Synchronization

B3: 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.

When we do directory operations we will first acquire `file_lock` that prevent other processes from accessing this inode. The lock will be released after all operations is done.

B4: 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?

Yes, we ignore all future file system operations relevant to it after is is removed.

Rationale

B5: Explain why you chose to represent the current directory of a process the way you did.

We choose to store the current thread's working directory in the struct of thread change it every time `chdir` is called, it is very convenient for us to just store it in each threads because we can easily access and modify it.

Task 3: Buffer Cache

Data Structures

C1: 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.

```
struct cache
{
  /* Whether this cacheline is free. */
  bool free;
  /* Dirty bit */
  bool dirty;
  /* The time_ticks last used, for LRU evict. */
  int64_t time_stamp;

  /* The corresponding sector */
  block_sector_t disk_sector;
  /* Data */
  uint8_t buffer[BLOCK_SECTOR_SIZE];
};
```

Algorithms

C2: Describe how your cache replacement algorithm chooses a cache block to evict.

We use LRU, whenever a cache is changed (used or created), we update the access time (saved as timestamp in cache) to the current timer ticks.

We use an array to record all the cache blocks.

When a cache block need to be evicted, we choose a least recently used (LRU) cache block by finding the longest access time, which means it is used earliest.

C3: Describe your implementation of write-behind.

Each time we want to read/write from sectors in disk, we always call `buffer_cache_read/write` instead to write/read from cache (a write operation will dirty the cache block).

Only when a dirty cache block need to be evicted, we will write it back to disk.

```
if (entry->dirty) {
    block_write (fs_device, entry->disk_sector, entry->buffer);
    entry->dirty = false;
}
```

C4: Describe your implementation of read-ahead.

Every time `buffer_cache_read`, the block of the sector will be copy to cache block, when another access tries to read/write the same block, it will read from cache block instead, which is faster.

Synchronization

C5: 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 `time_stamp` to store the last accessed time and evict the LRU one. Since the `time_stamp` of a buffer cache block will be updated whenever it's read/write, the mentioned cache block is not likely to be evicted when it is actively read/write because it's `time_stamp` is not likely to be the LRU one.

C6: During the eviction of a block from the cache, how are other processes prevented from attempting to access the block?

As mentioned, we have a lock `buffer_cache_lock` for all cache blocks. We need to acquire the lock when evicting the cache block and release after the eviction.

And at the same time, other processes is waiting this lock so they cannot access the block now.

Rationale

C7: Describe a file workload likely to benefit from buffer caching, and workloads likely to benefit from read-ahead and write-behind.

When we repeatedly access one small file for multiple times or we modify one large file using buffer caching will improve the efficiency by avoiding repeatedly access disk.

Read-ahead will benefit large file modification and write-behind will benefit both repeatedly small file access and large file modification.

Contributions

Cunhan You:

- Buffer Cache
- Syscall for Subdirectories
- Synchronization

Junda Shen:

- Indexed and Extensible Files
- Kernel Implementation for Subdirectories
- Robustness