CS 130: Project 4 - File Systems Design Document

Group Information

Please provide the details for your group below, including your group's name and the names and email addresses of all members.

- **Group Name**: *[printf("team")]*
- **Member 1**: Zirui Wang <wangzr2024@shanghaitech.edu.cn>
- **Member 2**: Xing Wu <wuxing2024@shanghaitech.edu.cn>

Preliminaries

> If you have any preliminary comments on your submission, notes for the TAs, or extra credit, please give them here.

None.

Indexed and Extensible Files

Data Structures

> **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.

Changed Struct to `filesys/inode.c`

```
struct indirect_block
{
  block_sector_t sectors[128]; /* Pointers to data blocks. */
};

struct inode_disk
{
  off_t length;                   /* File size in bytes. */
  block_sector_t parent;          /* Parent directory inode number. */
  block_sector_t direct[10];      /* Direct pointers to data. */
  block_sector_t indirect;        /* Indirect pointer to data. */
  block_sector_t doubly_indirect; /* Doubly indirect pointer to data. */
  unsigned magic;                 /* Magic number. */

  /* Not used. */
  uint8_t unused[BLOCK_SECTOR_SIZE - sizeof (block_sector_t) * 12
                 - sizeof (off_t) - sizeof (unsigned)];
};

struct inode {
```

```
  /* ... */
  struct rwlock rwlock;   /* Read-write lock for inode. */
};
```

New Struct to `threads/synch.h`

```
/* A fair read-write lock. */
struct rwlock
{
  struct lock lock;         /* Lock for mutex access. */
  struct list waiters;      /* List of waiting threads. */
  unsigned active_readers; /* Number of active readers. */
  unsigned active_writers; /* Number of active writers. */
};
```

New Struct to `threads/synch.c`

```
struct rwlock_waiter
{
  struct list_elem elem; /* List element for the waiters list. */
  bool is_writer;        /* True if this waiter is a writer. */
  struct semaphore sema; /* Semaphore for the waiter. */
};
```

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

About 8.07MB, every indirect block could contain 128 sectors, each sector contains 512 Bytes.

$$\frac{10 \times 512}{1024^2} + \frac{128 \times 512}{1024^2} + \frac{128 \times 128 \times 512}{1024^2} \approx 8.07MB$$

Synchronization

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

We designed a read-write lock which allows multiple reader or single writer accessing the critical section. If a file needs to be extended, a writer's lock needs to be acquired, which avoids two processes extending 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.

A acquires reader's lock while B acquires writer's lock, their operations won't be parallel.

- If A operates first, it will read none of what B writes.
- If B operates first, A will wait for B to finish, then read part or all that B writes.

A5: Explain how your synchronization design provides "fairness."File access is "fair"if readers cannot indefinitely block writers or vice versa–meaning that many readers do not prevent a writer, and many writers do not prevent a reader.

All request threads will be stored in a FIFO queue, and they will be handled in order.

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?

Our inode structure use a multilevel index, which is piratical in real-world cases:[1]

"Truths"(T)	Data (D)
Most files are small	Roughly 2K is the most common size
The average file size is growing	Almost 200K is the average size
Most bytes are stored in large files	A few big files use the most space
File systems contain lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain 50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Ten direct pointers can contain 5KB data, which makes random access for small files efficient.

One doubly indirect pointer makes storing large files possible.

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 inode_disk
{
  /* ... */
  int32_t is_dir;                 /* True if this inode is a directory. */
  int32_t file_cnt;               /* Only useful when it is a directory. */
  /* ... */
  uint8_t unused[BLOCK_SECTOR_SIZE - sizeof (block_sector_t) * 13
                 - sizeof (off_t) - sizeof (unsigned) - sizeof (int32_t) * 2];
}
```

[1] https://azrael.digipen.edu/~mmead/www/Courses/CS180/FileSystems-1.html

Algorithms

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

First we have a function to split the `char *name` into two parts: `path_name` and `file_name`. When traversing from absolute path, we will open the root dir; otherwise we will open the current working directory. After that, `dir_walk()` is called, using `strtok_r` to split the path by `'/'`, trying to find the token in directory and walking deeper.

Synchronization

> **B4:** 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.

If different directory entries pointes to the same inode, their operations are protected by the reader-writer lock inside each inode. Therefore, two simultaneous attempts to remove a single file won't be parallel. One attempts will succeed and the other will fail since the file does't exist. When it comes to creating simultaneously, one attempts will succeed and the other will fail since a file already exists.

> **B5:** 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?

We disallow this case. Before removing a directory, we check its sector is not the cwd of current process, and its inode's open count is not greater than one.

Rationale

> **B6:** Explain why you chose to represent the current directory of a process the way you did.

We choose to only store the `block_sector_t` instead of `struct dir *`. Some threads are created even before the file system is initialized, which makes the latter approach more tricky.

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.

New Struct Member to `filesys/cache.c`

```
/* Cache a sector of disk storage. */
struct cache_block
{
  struct block *block;             /* Pointer to the block device. */
  block_sector_t sector;           /* Sector number of the cache block. */
  bool dirty;                      /* true if block dirty, false otherwise. */
```

```
  bool valid;                     /* true if block valid, false otherwise. */
  uint8_t data[BLOCK_SECTOR_SIZE]; /* Data stored in the cache block. */
  struct lock lock;      /* A more fine-grained lock for this cache block. */
  struct list_elem elem; /* List element for the cache list. */
};

/* Read-ahead helpers. */
struct read_ahead_data
{
  struct list_elem elem; /* Element in read ahead list. */
  block_sector_t sector; /* Sector number of disk location. */
};
```

Algorithms

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

We use a Least Recently Used (LRU) replacement policy.

To implement this, we maintain a list of cache blocks ordered by their last access time. Whenever a cache block is accessed, we move it to the back of the list to mark it as most recently used. When eviction is required, we remove the block at the front of the list, since it is the least recently used.

C3: Describe your implementation of write-behind.

First maintain a `dirty` byte in each cache block. When a cache is evicted, dirty blocks will be written to the disk.

We also create a thread to periodically invoke `cache_flush()`, writing all dirty pages into the disk.

C4: Describe your implementation of read-ahead.

To implement this, we use a list and a semaphore.

First, create a read-ahead thread waiting for `sema_down()`.

In `inode_read_at()`, for each `offset` in the loop, send a read-ahead requirement at `offset + BLOCK_SECTOR_SIZE` to read next block asynchronously.

Read-ahead rquirements are pushed into a list, then the thread called `sema_up()`. The read-ahead thread will be wake up and get requirements in the list to read the block into cache.

List operations may cause race condition, so we use a lock to avoid potential problems.

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 a per-block lock to prevent this case. The reading/writing thread will acquire that block's lock first and other processes are not able to evict it.

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

We use a per-block lock to prevent this case. The evicting thread will acquire that block's lock first and other processes are not able to access it.

Rationale

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

If we are always reading and writing a small region of a file, those "hot"disk sections will be cached into memory, thus more efficient.

If we read a large contiguous chunks of file in order, it will likely to benefit from read-ahead.

If we keep logging small messages, there will be a lot `write()` system calls, and it will likely to benefit from write-behind.

Survey Questions

> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects? Any other comments?

None.