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**: *lorin&lucy*
- **Member 1**: Lufei Li `<lilf2024@shanghaitech.edu.cn>`
- **Member 2**: Jintong Luo `<luojt2022@shanghaitech.edu.cn>`

Preliminaries

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

No.

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.

```
#define INDIRECT_BLOCKS_PER_SECTOR 128
#define DIRECT_BLOCKS_COUNT 64

struct inode_disk
  {
    off_t length;                          /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    bool is_dir;
    uint32_t unused[59];                /* Not used. */

    block_sector_t direct_blocks[DIRECT_BLOCKS_COUNT];  /* Direct blocks. */
    block_sector_t indirect_block;                      /* Indirect blocks. */
    block_sector_t double_indirect_block;               /* Double indirect blocks.
*/
  };
```

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

*We have 64 direct blocks, 1 indirect block, and 1 double indirect block, which can store 64 + 128 + 128 * 128 data blocks, or (64 + 128 + 128 * 128) * 512 bytes.*

Synchronization

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

Writing to a file may cause file extention. We will first gain a lock on the file in `sys_write` system call.

> **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.

In `sys_read` and `sys_write` system call, we will first gain a lock `cache_lock` on the file, and then operte on the file. So there will be no race condition.

> **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.

Every file-related syscall will gain a lock first, specifically `sys_read` and `sys_write`, and release the lock immediately after they finish the operation. This ensures that neither readers nor writers can indefinitely block each other, providing a fair access mechanism.

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 is a 3-level index with direct, indirect, and double indirect blocks. We chose this combination to balance between space efficiency and performance. The direct blocks allow for fast access to small files, the indirect block extends the file size while keeping the inode size manageable, and the double indirect block further extends the file size to support files larger than 8MB. This structure provides a good trade-off between the number of disk accesses required to access a file and the maximum file size supported.

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.

The original structure of dir is enough, so we didn't add any new struct or struct member.

```
/* A directory. */
struct dir
{
    struct inode *inode;                /* Backing store. */
    off_t pos;                          /* Current position. */
};

/* 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? */
```

```
  };
```

Algorithms

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

We first split the name into directory path and file name, and then traverse the directory path to find the inode of the directory.

For absolute paths, it starts from the root directory, while for relative paths, it starts from the current working directory of the process.

It then iteratively opens each directory in the path and looks up the next component until it reaches the final component. If any directory in the path does not exist or cannot be opened, the traversal fails.

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.

By accquiring a lock before directory operations.

> **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?

Yes, it is allowed to remove a directory if it is open by a process. But it cannot be opend after it is deleted.

So nothing dangerous will hapen in future operations.

Rationale

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

We chose to represent the current directory of a process using a pointer to a dir struct. This allows us to easily keep track of the current working directory and perform operations on it, such as opening files and changing directories.

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.

```
/* Lock for synchronizing cache operations. */
struct lock cache_lock;

struct cache
{
    bool valid;                         /* Valid bit */
    bool dirty;                         /* Dirty bit */
    int64_t last_used;                  /* Last used time */
    block_sector_t sector;              /* Block sector */
    uint8_t buf[BLOCK_SECTOR_SIZE];     /* Buffer */
} cache_list[64];
```

Algorithms

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

Our cache replacement algorithm uses a least recently used (LRU) policy to choose a cache block to evict. When a new block needs to be loaded into the cache and there are no free blocks, the algorithm selects the block that was least recently used and evicts it, making room for the new block. The `find_available` function iterates through the `cache_list` to find the block with the earliest `last_used` *timestamp. If a block is found, it is evicted by writing its contents back to disk if it is dirty, and then it is marked as invalid.*

> **C3:** Describe your implementation of write-behind.

Our implementation of write-behind delays the writing of modified cache blocks to disk until the block is evicted from the cache or the cache is explicitly flushed. When a block is modified, it is marked as dirty.

> **C4:** Describe your implementation of read-ahead.

The cache mechanism inherently supports read-ahead by caching blocks that are read from disk. When a block is read, it is stored in the cache, and subsequent reads of the same block can be served from the cache without accessing the disk.

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?

The `cache_read` and `cache_write` functions acquire the `cache_lock` before accessing or modifying the cache block. This ensures that only one process can access the cache block at a time.

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

During the eviction of a block from the cache, other processes are prevented from accessing the block by using `cache_lock` on the cache.

Rationale

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

Buffer caching improves performance by reducing the number of disk accesses required.

Workloads that benefit from read-ahead include sequential file access patterns, such as reading a large file from beginning to end. Read-ahead reduces the number *of disk reads required by prefetching blocks in advance.*

Workloads that benefit from write-behind include applications that perform frequent small writes to a file.

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?

No.