```
+-------------------------+
|         CS 140          |
| PROJECT 4: FILE SYSTEMS |
|     DESIGN DOCUMENT     |
+-------------------------+
```

---- GROUP ----

> Fill in the names and email addresses of your group members.

Yining She sheyn@shanghaitech.edu.cn

Shaoting Peng pengsht@shanghaitech.edu.cn

---- PRELIMINARIES ----

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

> Please cite any offline or online sources you consulted while
> preparing your submission, other than the Pintos documentation, course
> text, lecture notes, and course staff.

Online sources: https://wenku.baidu.com/view/5a71dfd232d4b14e852458fb770bf78a65293aa8.html?re=view

Offline sources: pintos guide

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.

In `inode.c`:

```
#define INODE_TABLE_LENGTH 128
#define INODE_DIRECT_N 8
#define INODE_INDIRECT_N 32

struct inode_disk
  {
    ...
    // direct blocks
    block_sector_t direct_blocks[INODE_DIRECT_N];
    // indirect blocks
    block_sector_t indirect_blocks[INODE_INDIRECT_N];
    // whether it's a dir
    bool is_dir;
    ...
  };
```

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

As shown above, we have 8 direct blocks and 32 indirect blocks (according to the discussion session provided by our kind TA), so the max size of a file supported is:

8 * 512 + 32 * 128 * 512 = 2 MB

P.S. The 8 direct blocks are used for fast access a file.

---- SYNCHRONIZATION ----

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

Two processes are not able to extend a file at the same time in our design. To access a file, the process must raise a system call, and according to our implementation, every single syscall is given a lock to ensure its atrocity, so two processes can't access the same 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.

Similar as above analysis, by rising **write** and **read** syscall, we asure its atomicity. So, if process A has a priority higher than process B, A would read before B writes anything. On the other hand, if process B's priority is higher, A would read the data writen by B, so there will not be any races.

> 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.

Our implementation didn't explicitly consider this problem. But readers and writers are both processes, so they are in a queue according to their priorities. If they have the same priorities, they are treated equally in the waiting list, and thus "fairness" is ensured.

---- 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 implementation of inode structure have a multilevel index, which is 8 direct blocks and 32 indirect blocks.

We choose 32 indirect blocks because TA Yanjie Song, a very kind TA, told us that 2MB of a file is enough in the discussion session;

We choose 8 direct blocks because they can accelerate accessing a file.

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.

In `thread.h`:

```
struct thread
{
  ...
  // current directory
  struct dir *pwd;
  ...
}

struct file_node
{
  ...
  // directory pointer associated with the file
  struct dir* dir_ptr;
  ...
};
```

---- ALGORITHMS ----

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

We divide the path into directory and file name using the following function:

```
bool dir_divide (char *name, struct dir *cur_dir, struct dir **ret_dir, char
**ret_name)
{
  ...
}
```

The input is name, which is the user-specified path, and current directory. The output is specified as `ret_dir` and `ret_name`. If it starts with a '/', then we start from root, otherwise we start from `cur_dir`, which is the current directory. Then, we use the following `for` loop to traverse the path:

```
for (token = strtok_r (path, "/", &save_ptr); token != NULL; token = strtok_r (NULL,
"/", &save_ptr))
{
  ...
}
```

Traversal of absolute and relative paths behaves similar, because we regard current directory and '.' the same, so we only need to use a `if` sentence to check for absolute or relative path.

---- 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, and so on.

The newly implemented system call have the same lock as before, which means every directory operation is ensured to be atomic, consequently the races on directory entries are prevented.

> 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?

Our implementation didn't allow a directory to be removed while open.

We have a struct variable in `inode`: `open_cnt`, which count for how many processes is currently with this file open. If `open_cnt` is larger than 1, we return false in `filesys_remove()`.

---- RATIONALE ----

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

We represent the current directory in each thread, so we add a struct variable called `pwd` to represent it. When a process is running, this directory keeps open. When it finishes, close the directory as well. In this way, it would be much easier to maintein.

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.

In `cache.c`:

```
/* the struct of cache entry */
struct cache_sector
{
    // cache buffer
    unsigned char buffer[BLOCK_SECTOR_SIZE];
    // lock for EACH cache sector
    struct lock cache_lock;
    // current cache's sector index
    block_sector_t sector_id;
    // how many time this cache is accessed
    int accessed;
    // dirty bit
    bool dirty;
    // whether it's used
    bool used;
};

/* the cache, which is an array of struct cache_sector */
static struct cache_sector cache[64];
```

```
    /* lock used for the whole cache operations */
    static struct lock cache_big_lock;

    /* current cache pointed. Used for clock algorithm */
    int cache_cur;
```

---- ALGORITHMS ----

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

We choose **clock algorithm** as our cache replacement.

We maintain a variable `cache_cur` to store the current cache block the clock is pointing to, and we implement the following `while` loop:

```
    int fetch_free_cache ()
    {
        // use clock to find a cache entry to evict
        while (true)
        {
            if (cache[cache_cur].used == false) {
                int temp = cache_cur;
                cache_cur = (cache_cur + 1) % 64;
                return temp;
            }
            if (cache[cache_cur].accessed == 0){
                if (cache[cache_cur].dirty == true)
                    block_write (fs_device, cache[cache_cur].sector_id,
    cache[cache_cur].buffer);
                cache[cache_cur].used = false;
                int temp = cache_cur;
                cache_cur = (cache_cur + 1) % 64;
                return temp;
            }
            cache[cache_cur].accessed = 0;
            cache_cur = (cache_cur + 1) % 64;
        }
    }
```

We first check the **used** bit. If not used, return it directly;

Then we check the **accessed** bit, which give it a second chance. If `cache[cache_cur].accessed = 0` already, we choose this cache block to evict; else we set it to 0.

Notice that in every case, we move the `cache_cur` to point to the next cache block.

> C3: Describe your implementation of write-behind.

We implement a function called `write_behind()` and call it in `cache_init()`. `write_behind()` function creates a new thread with thread function `write_behind_func()`, which is a simple function:

```
void write_behind_func ()
{
    while (true){
        // flush back every 0.5s
        timer_msleep (500);
        cache_back_to_disk ();
    }
}
```

In this way, cache entries are pushed back to disk every 0.5s.

> C4: Describe your implementation of read-ahead.

We implement a function called `read_ahead()` and call it in `cache_init()`. `read_ahead()` function creates a new thread with thread function `read_ahead_func()`, which waits for a condition variable `read_ahead_condition` on whether a list, `read_ahead_list`, which contains the next element of the current cache (in `cache_read()` or `cache_write()`), is not empty anymore, and pop the first element in `read_ahead_list`. Then, find the `cache_id` according to the `sector_id`, and invokes `fetch_free_cache()` if this sector is not in cache yet.

---- 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?

In each cache function (i.e. `cache_read()` and `cache_write()`), we have a lock called `cache_big_lock`, initiated in `cache_init()`. Whenever a cache operation of a process A started, we acquire this lock, and release this lock only when the cache operation finishes. In this way, during a cache operation, process B can't access to the caches process A is currently opearting.

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

Similar as above, out function to evict block, `fetch_free_cache ()`, is only invoked in a cache function (i.e. `cache_read()` and `cache_write()`), so we can ensure its atomicity by acquire and realease `cache_big_lock`, as introduced before.

---- RATIONALE ----

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

If a process keeps **reading and writing to the same blocks**, the file workload would benefit a lot from buffer caching.

As for "read_ahead", the file workload would benefit when a process tries to access the file blocks **sequentially**.

If the system is **not stable enough**, or the process keeps many files **always open**, the file workload would benefit.

SURVEY QUESTIONS

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

> > In your opinion, was this assignment, or any one of the three problems
> > in it, too easy or too hard? Did it take too long or too little time?

> Did you find that working on a particular part of the assignment gave
> you greater insight into some aspect of OS design?

> Is there some particular fact or hint we should give students in
> future quarters to help them solve the problems? Conversely, did you
> find any of our guidance to be misleading?

> Do you have any suggestions for the TAs to more effectively assist
> students in future quarters?

> Any other comments?