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

---- GROUP ----

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

Group 19
韩嘉恒 hanjh1@shanghaitech.edu.cn
李鹤洋 lihy3@shanghaitech.edu.cn

Contributions:
Discuss together, and coding part:
Indexed and Extensible Files by Li Heyang,
Subdirectories, work together,
Buffer cache, by Han Jiaheng.
---- 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.

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.

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
    //block_sector_t start;             /* First data sector. */
    block_sector_t blocks[12];          /* Block array for extending files. */
    uint32_t direct_usage;             /* Record the number of used direct
blocks. */
    uint32_t indirect_used;            /* Record whether the indirect block is
used. */
    uint32_t indirect_block_usage;      /* Record the number of used indirect
blocks. */
    uint32_t double_used;              /* Record whether the doubly-indirect
block is used. */
    uint32_t double_l1_usage;           /* Record the number of used level-1
doubly-indirect blocks. */
    uint32_t double_l2_usage;           /* Record the number of used level-2
doubly-indirect blocks. */
    uint32_t sector_usage;             /* Record the number of total sectors
used. */
    off_t length;                     /* File size in bytes. */
    unsigned magic;                   /* Magic number. */
    bool is_dir;                      /* Determine whether the inode is file or
directory. */
    block_sector_t parent;             /* Record the parent directory of this
inode_disk. */
    uint32_t unused[105];              /* Not used. */

  };
```

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

We implement a 12-block structure, with 10 of them are direct blocks,
1 is indirect block and 1 is doubly-indirect block.
One direct block has 512 bytes (1 sector),
One indirect block has 128 direct blocks,
And one doubly-indirect block has 128 doubly-indirect blocks.
So total size of a file has 512*(1+128+128*128)=8454656 bytes.

---- SYNCHRONIZATION ----

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

Because we acquire the lock before writing to files and release it after the writing, there can only be 1 process writing to the file at the same time. This means two processes can’t attempt to write extend the file in a race. So we avoid the race by synchronization.

>> 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 our implementation, this depends on which operation comes first. Because we use a lock to prevent this happens. If A reads before B writes, A reads no data, elsewise B writes first and A reads all data that B writes. Our read and write are atomic operations. Read and write can’t happen at the same time, so this kind of race is solved by external 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.

Our design naturally meets the requirement of this “fairness”. Because in our implementation, write and read operations are like in a FIFO queue, the one comes first starts its operation until all the work are done, then the next operation follows and do what it should do. Read are write are atomic operations here so no operation can forever prevent another process read or write.

---- 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?

We use a multilevel index with 10 direct blocks, 1 indirect block and 1 doubly-indirect block. We choose this structure because it' s a simple way to meet the requirement of the size of file and easy to implement. We are flexible to modify the number of different level blocks and maximum size we want and it' s friendly to space.

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

```
   /* Project4 */
   struct dir* cur_dir;             /* Current directory. */
```

---- ALGORITHMS ----

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

We use strtok_r () to divide the path into different levels. We then start from the root directory if the path is an absolute path or the working directory if the path is a relative path. This can be told whether the path start with a '/'. For each string we get from strtok_r (), if we can find the directory with the name in the current directory, we go into it, otherwise return false, which meanings the path is not valid. And we keep go on until we finish the path and return the final inode.

---- 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.

Because our remove or create are atomic operations, which means two removing or two creating can' t happen at the same time, only when one is done, another one can be processed. So if one create or remove operation is successfully done, another attempt would fail and return because the file has already been created or removed. Then the system works as usual.

>> 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 don' t allow a directory to be removed if it is open by a process or it is in use as a process's current working directory. Before removing the directory, I would check if any process has opened this directory or in use as it' s current working directory, if so, I deny the operation of removing it and work as usual.

---- RATIONALE ----

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

I store the current directory as struct* dir in struct thread. Because each process has a working directory so the struct thread is the best choice. And struct* dir is unique to identify, and it' s easy to use at the same time. If we store the name of dir, it' s name can be change, so we can easily lose track of it. Using struct dir* is more stable and reasonable.

                 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.

```
#define CACHE_SIZE 64
#define READ_AHEAD_BUFFER_SIZE 64

struct cache_block
  {
    /* Whether the cache line is dirty.*/
    bool dirty;
    /* Whether the cache line is valid.*/
    bool valid;
    /* The cooresponding block sector on disk. */
    block_sector_t disk_sector;
    /* Most recently use time. */
    int64_t time;
    /* Semaphore for this cache line. */
    struct semaphore sema;
    /* Which block(device), in this project, always fs_device.
*/
    struct block *block;
    /* The cached BLOCK_SECTOR_SIZE size date. */
    char disk_data[BLOCK_SECTOR_SIZE];
  };

/* Pointer to the filesystem cache. */
static struct cache_block *cache;

/* Information for a single read-head operation. */
struct read_head_elem
  {
    block_sector_t sector;
    struct block *block;
  };

/* We use "producer" and "consumer" pattern to handle read-
head. */
/* All variable for read-head function. */
static struct read_head_elem *read_head_buffer;
static struct condition read_head_buffer_not_full;
static struct condition read_head_buffer_not_empty;
static struct lock read_head_lock;
static uint32_t read_head_buffer_n;
```

---- ALGORITHMS ----

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

LRU is used as the cache replacement algorithm. Each cache line has a variable `time` to record its most recent access time and a variable `valid` to record whether it is valid. And when do evict, we will go through all cache line, if any invalid cache line is found, it is chosen as evict, otherwise the cache line with the earliest most recent access time will be evicted.

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

When we try to write a block to disk, we will write it to cache, and set the dirty bit to true. When a valid cache line is evicted, if it is dirty, the data will be written back to disk. Also, when the system is shutdown, we will flush the cache, and we create a thread to periodically flush the cache to enhance reliability.

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

We use "producer" and "consumer" pattern to handle read-head. Each time when we have a read operation, we will produce a read-ahead operation, stores which block sector to read into cache by put_read_ahead_buffer ()and store it in read_head_buffer. We will also create a thread read_ahead ()to handle the read-ahead operation, where read-ahead can be done asynchronously.

---- 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?

Each cache line has its own semaphore sema. And before we try to read, write, find, or evict a cache line, we will always call sema_down () on its semaphore, which prevents data race.

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

In choose_evict (), before we check, modify a cache line, we will first call sema_down () on its semaphore, then check whether it should be chosen as the evict. Each time we choose a new evict in the process of LRU, we will call sema_up () on the old evicted cache line' s semaphore. The caller of choose_evict () should be responsible to call sema_up () on the evicted cache line' s semaphore so the whole process on the single cache line is atomic.

---- 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 filesystem’ s resident usage is lower than the cache size and the resident usage is repeatedly accessed, it can benefit from the buffer caching. Files that are read sequentially and stored sequentially on disk are more likely to benefits from read-ahead and files that have a certain place that are repeatedly written are more likely to benefit from write-behind.

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?