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

---- GROUP ----

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

Zhaozheng Shen <shenzhzh@shanghaitech.edu.cn>
Wenhui Qiao <qiaowh@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.

CSCI 350: Pintos Guide Written by: Stephen Tsung-Han Sher

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.

1. global or static variable:

```
#define DIRECT_BLOCK 120      /* An inode has DIRECT_BLOCK direct
entries. */
#define INDIRECT_BLOCK 128         /* An inode has a INDIRECT_BLOCK
entry. */
#define DOUBLE_INDIRECT 128 * 128 /* An inode has a DOUBLE_INDIRECT
entry. */
```

2. changed `struct' member

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
    block_sector_t direct_part[DIRECT_BLOCK];  /* direct part of an
inode. */
    block_sector_t indirect_part;           /* indirect part of an
inode. */
```

```
    block_sector_t double_indirect_part; /*double direct part of an
inode. */

    // block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    // uint32_t unused[125];               /* Not used. */
    block_sector_t parent;  /* store parent */
    bool dir_or_file;/* store the type of the inode(ordinary or
directory). */
  };


3. new `struct'
/* the struct of the indirect part stored in one inode. */
struct inode_indirect
{
  block_sector_t indirect_inode[INDIRECT_BLOCK];  /* indirect inode
entry. */
};
```

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

As showed above, an inode is composed by 120 direct blocks, 1 indirect block, and 1 double indirect block, so the whole number of block is 120 + 1*128 + 1*128*128 = 16632 blocks. And each block is 512 byte, so an inode contains 16632*512 = 8515584 byte, which is 8515584/(1024*1024) = 8.12109375G.

---- SYNCHRONIZATION ----

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

Since there is a lock before trying to extend a file, only one process can extend a file at a time, and only one hold the lock, so there is no way for another process to extend the file.

>> 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.

Only after process B start to write a new section, will it update its file length. So if process A read the updated length that process B write, it will always read data written by B.

>> 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.

One process cannot always be running since our operating system will block the process that is running every some time ticks. In this way, a process cannot indefinitely block writers or vice versa.

---- 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, our inode is a multilevel index, it has 120 direct blocks, one indirect block and one doubly indirect block. Since we have read the pintos guide written by Stephen Tsung-Han Sher, and the figure of inode structure in his guide is direct + indirect + double indirect, so we just use the structure.
The TA said that we can only file all 128 entries with all indirect blocks, but we still use our design. The advantage is that many files are small and it only just stored in direct block, it is much faster to access the direct blocks, so accessing these files is faster than all indirect blocks design.

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.

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

```
/* Structure for entry in the directory. */
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?

When we are going to traverse a user-specific path, the first thing we need to do is to check the first character of the path. If it is '/', it means it is the absolute path, so we need to open the root directory. Otherwise, it's relitive path so we need to open the current working directory, which is a member variable in the struct of thred.

Once we have opened a directory, we are going to search the next entry in the directory. If entry with the given name is found, then search it for the next entry. Otherwise, stop and return.

---- 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.

On one hand, when we going to create the file or delete it, we need to acquire the lock. That prevent from data race in dircetory entry. On the other hand, only one of two simultaneous attempts to selete the file woule succeed, as the other one will not be able to find the file.

>> 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?

No. During the process of removing a diretory, we will check whether this directory entry is in use or not, which is represented by a boolean value in the structure of dir_entry. So if it is in use, we will not remove the directory.

---- RATIONALE ----

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

In the structure of thread, there is a pointer to the directory called cwd, which represent the "current working directory". Such

method would be easy to implement, and a pointer in the struct would not use much of memory space. Also it is convenient to update.

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.

Structures as below:

```
/* Entry in the cache array. */
struct cache_entry {
    char data[BLOCK_SECTOR_SIZE];   /* Data buffer. */
    struct lock entry_lock;         /* Lock for this entry. */
    block_sector_t sector;          /* Sector number of the data. */
    bool valid;                     /* Whether initialized. */
    bool dirty;                     /* Modified or not. */
    bool accessed;                  /* Been read or not. */
};

/* Entry for read ahead. */
struct entry_read {
    struct list_elem elem;          /* Elem for put in list. */
    block_sector_t sector;          /* Sector number. */
};
```

static variables:

```
/* Tools for synchronization. */
static struct lock cache_lock;          /* Lock for whole cache. */
static struct list read_ahead_queue;    /* Queue for read ahead. */
static struct lock ahead_lock;         /* Lock for read ahead. */
static struct condition ahead_cond;     /* Condition for read_ahead. */

/* Cache body. */
static struct cache_entry cache[CACHE_ENTRY_NUM];   /* Array of the
cache. */
```

---- ALGORITHMS ----

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

The general idea of eviction is clock algorithm. Iterate through the cache, for each entry, check whether it has been accessed. If it has been accessed, give it second chance, which means set the bool value of accessed to false. If it has not been accessed, evict this entry.

There are other factors that need to be considered. One is that trying to aquire the entry lock. If fail, that means other process is trying to read or write the entry. As a result, we should skip this entry and move down to the next entry.

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

On one hand, every time we need to evict an entry from cache, if the entry is dirty, we need to write the data back to disk, which is achieved by calling the function of "block_read".

On the other hand, we defined a working function that iterate through the whole cache, if it is dirty, write it back to disk, and then set the dirty to false. Also, we need to define another function that call the previous function periodically. And during the initialization, we creat the new thread with such function.

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

Create a list of struct entry_read, which include the sector that need to be read in the process of read-ahead.

If there is one sector need to be read, it will allocate a new entry, set the parameters and push it into the list. Then another function is going to pop the entry from the list, and perform block_read.

If the list is temporaly empty, the thread will be waiting for a cindition variable, whose singal will be sent by another function after pushing the item into the list. We also create a thread with such function to achieve read-ahead.

---- 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 the structure of cache entry, there is a lock called entry_lock, which is used to prevent such circumstances. In the process of eviction, we need to perform a lock_try_acquire of the entry_lock. Failing to do so means that there is another thread which is performing read or write to this entry. And in this way, we will skip this entry and move down to the next entry.

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

The lock entry_lock could also solve this problem. When the ecivtion
process determined which entry to evict, it must have acquired the entry_lock of the entry.

In the process of read or write, when we call the function of find or bring in (i.e. either cache hit or miss), it will also acquire the lock of entry_lock. If not able to do so, it means it is being evicted. So it will wait until the ecivtion to finish.

---- 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:

It takes much more time to get to the disk and read or write the data we need, so we put part of the data which is likely to acccessed in the future in the RAM, which has a better speed of access. In this way, the whole file system could geta better performence in terms of average response time.

Write-behind:

If a sector from the disk is read, then it is very likely that the next sector be read. If we do not perform the read ahead, then it will have to wait for quite a long time to wait until the disk rotate a circle. So if we perform read ahead, it will improve the overall performence a lot.