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

---- GROUP ----

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

Yi Qian <qianyi1@shanghaitech.edu.cn>
Guancheng Li <ligch@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
https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/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 "filesys/inode.c":

```
struct inode_disk
{
    block_sector_t blocks[12];    /* 10: 1: 1 = Direct: Indirect: Double */
    off_t length;                 /* File size in bytes. */
    unsigned is_dir;              /* Is this inode a directory. */
    unsigned magic;               /* Magic number. */
    uint32_t unused[113];         /* Not used. */
};

/* In-memory inode. */
struct inode
{
    struct list_elem elem;        /* Element in inode list. */
    block_sector_t sector;        /* Sector number of disk location. */
    int open_cnt;                 /* Number of openers. */
    bool removed;                 /* True if deleted, false otherwise. */
    int deny_write_cnt;           /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;       /* Inode content. */
```

```
};

/* A struct to represent indirect block. */
struct indirect_block
{
   block_sector_t direct_blocks[128];
};

/* A struct to represent double indirect block. */
struct double_indirect_block
{
   block_sector_t indirect_blocks[128];
};
```

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

I have defined the size.

```
#define DIRECT_BLOCK_NUM      10
#define INDIRECT_BLOCK_NUM    1
#define DOUBLE_INDIRECT_NUM   1
#define DIRECT_BLOCK_SIZE     (BLOCK_SECTOR_SIZE * DIRECT_BLOCK_NUM)
#define INDIRECT_BLOCK_SIZE   (BLOCK_SECTOR_SIZE * DIRECT_PER_INDIRECT)
#define DOUBLE_INDIRECT_SIZE  (INDIRECT_PER_DOUBLE * INDIRECT_BLOCK_SIZE)
```

So the sum of the three parts is:

```
512 * 10 + 512 * 128 + 512 * 128 * 128 = 8459264 B
                                  = 8.07 MB
```

---- SYNCHRONIZATION ----

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

If two processes want to create a file at the same time. They both need to create an inode. There is a lock called lock_of_open_inodes to ensure that only one process can create an inode or remove an inode.

If they extend a file by writing. The extending process completes by the cache.

So we add a lock to ensure that only one process can do operations to enter the buffer cache and each cache block has a lock to ensure the synchronization.

So it can avoid a race.

>> 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 inode_disk, their is a member variable called length which represents the file's current length.

The length will be updated only when whole writing process is finished. So if B is writing, A is reading, A won't know that the file's length is extended.

And inode_read_at() use the length to determine how many bytes it can read.

So our code can avoid the race.

>> 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.

We allow readers and writers to do operations on a same file. But readers and writers can't get access to the same data block in the same time. It means that different readers and writers can read or write the same file at the same time as long as not the same data block in the file.

---- 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, out inode structure a multilevel index.

We use 10 direct blocks, 1 indirect block and 1 doubly indirect block.

The reason why we choose this design is that this design can satisfy the file's largest size restriction (8MB) and small file will benefit a lot because of there is some direct blocks and indirect block.

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.

Nothing new.

---- ALGORITHMS ----

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

Given a user-specified path, we first check if it starts with a '/'. If true, the path is an absolute path and we open the root directory to begin with. If not, we reopen current directory to begin with.

We store a '.' entry and a '..' entry in every directory, so we will not treat these symbols differently than real names. We parse the path with '/' and at each step go through current directory to find an entry matching the name. If the entry we find denotes a directory, we move our position into that directory.

Traversals of absolute and relative paths differ only in the beginning, we choose a different place to start with.

---- 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.

There is a lock when changing an inode. Doing operations on directory entries results in changing the inode behind it, and the lock do not allow two simultaneous changes to the same inode.

>> 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 do not allow.

When the process asks to remove a directory, we check if it is opened by a process and if is in use as current working directory. If true, we reject the remove request.

---- RATIONALE ----

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

We store the pointer of current directory (struct dir *) in the thread struct. Struct thread is unique to a process, holding the current directory in there is appropriate. Also, current working directory should be kept open until the process changes its working directory. The directory pointer must be kept to close it after it is no longer in use.

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.

```
struct cache_block
{
    uint8_t data[BLOCK_SECTOR_SIZE]; /* One block contains 512 bytes data. */
    block_sector_t sector_idx;       /* Data from sector_idx in disk. */
    bool used;                      /* Whether this cache block is used. */
    bool dirty;                      /* Dirty bit. */
    bool accessed;                  /* Access bit, for clock policy. */
    struct lock lock_of_cache_block;   /* Single cache block lock. */
};
```

---- ALGORITHMS ----

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

We use the clock policy.
In each cache block entry, there is a member variable call accessed.
Each write or read operation related to this block will make the variable become true.

When we need to evict a cache block, we start from the cache block clock hand pointing to and check whether the variable accessed is true. If it is true, make it false and check the next block until we find a cache block the variable accessed is false. And if it is dirty, write it back. Finally, evict this block. The policy will check the buffer cache round by round until finding a block.

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

```
void
cache_write_back_all ()
{
   struct cache_block* block = NULL;
   lock_acquire (&lock_of_buffer_cache);

   for (int i = 0; i < CACHE_BLOCK_NUM; i++)
      {
      block = buffer_cache[i];
      lock_acquire (&block->lock_of_cache_block);

      if (block->used && block->dirty)
         {
         block_write (fs_device, block->sector_idx, block->data);
         block->dirty = false;
         }

      block->accessed = true;
      lock_release (&block->lock_of_cache_block);
      }

   lock_release (&lock_of_buffer_cache);
}

static void
cache_write_behind (void *aux UNUSED)
{
   while (true)
   {
      timer_sleep (PERIODIC_WRITE_TIME);
      cache_write_back_all();
   }
}
```

We create a thread to run the cache_write_back_all () function. The thread will write back all dirty blocks periodically. The period is defined by PERIODIC_WRITE_TIME.

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

```
struct read_ahead_elem
{
   block_sector_t sector_idx;         /* Disk sector_idx to read. */
   struct list_elem elem;             /* List elem. */
};
```

```
void
cache_add_read_ahead_elem (block_sector_t sector)
{
   struct read_ahead_elem *read = malloc (sizeof (struct read_ahead_elem));
   read->sector_idx = sector;

   lock_acquire(&lock_of_read_ahead);
   list_push_back(&read_ahead_list, &read->elem);
   cond_signal(&read_ahead_cond, &lock_of_read_ahead);
   lock_release(&lock_of_read_ahead);

}

/* Read ahead. */
static void
cache_read_ahead (void *aux UNUSED)
{
   struct read_ahead_elem *read;

   while(true)
   {
      lock_acquire (&lock_of_read_ahead);

      while(list_empty (&read_ahead_list))
      {
         cond_wait (&read_ahead_cond, &lock_of_read_ahead);
      }

      read = list_entry(list_pop_front(&read_ahead_list),
                  struct read_ahead_elem, elem);
      lock_release(&lock_of_read_ahead);

      cache_read_one_block(read->sector_idx, NULL);
      free(read);
   }
}
```

We also create a new thread to run cache_read_ahead () function. When the read_ahead_list is empty it cond_wait() until other threads signals it that the list isn't empty.
The read_ahead_elem will record the sector index to be read and it will be add to the read_ahead_list () by cache_add_read_ahead_elem () function. In this function, it signals the read_ahead thread let the thread read the sector
from disk to cache.

---- 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 block has a lock. If one process want to change the data or info

in a cache block. It must acquire the block's lock first. So it can ensure that the block won't be evicted if it is being used by another process.

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

As mentioned above, if one process is doing some operation on the block. It will acquire the block's lock first. So if one process is evicting a block, other processes won't get access to the block.

---- RATIONALE ----

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

If some file's data sector are accessed repeatedly, system can get the data from memory instead of from disk. It benefits the workload a lot.

If process do some operations on a file sequentially. Read ahead will make sure half data will be read into data before the process need them.

If a sector will be write back to disk frequently, write behind will make he main process save much time.

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?