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

---- GROUP ----

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

Yuxin Miao <miaoyx@shanghaitech.edu.cn>
Fan Zhang <zhangfan5@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.
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.

---------------- filesys.c ------------------

```
#define SECTOR_PTR_CNT (BLOCK_SECTOR_SIZE / sizeof(block_sector_t))
/* Number of meta data. */
#define META_PTR_CNT (SECTOR_PTR_CNT - BLOCK_PTR_CNT)
/* Number of data sectors. */
#define BLOCK_PTR_CNT 12
/* Number of indirect data sectors. */
#define INDIRECT_BLOCK_CNT 1
/* Number of double indirect data sectors. */
#define DOUBLE_INDIRECT_BLOCK_CNT 1
/* Number of direct data sectors. */
#define DATA_BLOCK_CNT (BLOCK_PTR_CNT - INDIRECT_BLOCK_CNT -
DOUBLE_INDIRECT_BLOCK_CNT)
/* Max length of inode, in bytes .*/
#define INODE_MAX_LENGTH ((DATA_BLOCK_CNT + \
                           SECTOR_PTR_CNT * INDIRECT_BLOCK_CNT + \
                           SECTOR_PTR_CNT * SECTOR_PTR_CNT *
DOUBLE_INDIRECT_BLOCK_CNT) * \
                          BLOCK_SECTOR_SIZE)

struct inode_disk
{
  block_sector_t sectors[BLOCK_PTR_CNT]; /* Sectors. */
  off_t length;                          /* File size in bytes. */
  int is_dir;                            /* File : 0 ; dir : 1 */
  unsigned magic;                        /* Magic number. */
  uint32_t unused[META_PTR_CNT - 3];
};

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. */
  struct lock inode_lock; /* Lock used for directory. */

  int being_written;            /* 0: writes ok, >0: deny writes. */
  struct lock writing_lock;     /* Lock for deny write. */
  struct condition writer_cond; /* Condition indicating no writers. */
  int writers;                  /* Can only deny write when there's no writer.
*/
};
```

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

In our disk inode, we use an array of pointers to take place of previous implementation. In our array we have 10 direct data pointers, 1 indirect pointer and 1 doublly_indirect pointer.
There are BLOCK_SECTOR_SIZE/sizeof(block_sector_t) = 128 pointers in an inode_disk struct.
So we can support 10*512+128*512+128*128*512 = 8459264Bytes which aproximately is 8MB.

---- SYNCHRONIZATION ----

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

In our implementation, a process can extend a file in an unit of cache line which means even a process wants to extends a lot of bytes beyond the EOF, it can only accomplish this cache line by cache line or one disk_inode in other words.

We give every cache and inode a lock, so when any process wants to access a cache line
it has to acquire this lock first. Then we do a condition seperations, if this is an
operation except for extension, which as described in the documentation, should all
multiple processes do work simultaneous, then we wake all the process waiting for this
cache line before we do any work. Otherwise, if this is an extension operation, we will
hole the lock until we finish the extension.

Further, we have a double check for extension. It only triggers extension only when the
belonging functions is called by the write_at functions and detects the sector that we
are intending to write has nothing in. If only above should we allocate a new place for
this sector and set zeros. So we double check the sector we are going to allocate at if
the sector is still blank and waiting for allocation.

>> 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.

We seperate the actual read functions on block_sectors into a seperate function

```
called read_block, only when the write_at calls this function do we allow the
process to do operation beyond EOF, in this case, it's extension zeros and write
new data into the sector if any. So if a read_at function calls the read_block
we return upon EOF so that process-A will not read anything except NULL.

>> 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 read data in unit of cache_line, that means if a files is seperated into
multiple inode_sector to store, only the one that is required will have its
lock acquired by the caller function, and the rest cache line is free to do
any modification.

Also, we only hold lock for a cache line thoughout extension operation, at
any other we will release the lock so that any one who want to read or write
can gain the lock and do their operations.

TO BE MORE CLEAR: As mentioned above, the lock on cache_line is acquired every
time a process access this cache line, between finishing acquiring the cache
line
and do the desired operation, we check the operation, if it's not extension, we
relase the lock immediately before operation.

---- 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 we use a multilevel index of 3 levels. We have 10 direct data pointers, 1
indirect
pointer and 1 doublly_indirect pointer.  So we can support aproximately 8MB.

As suggested on the slides, the combination is 12+1+1 so that we believe a
10+1+1
combination is enough for the pintos implementation. This combination is also
recommended in the USC pintos guide of their course project.

We also prepared to extend the number of indirect data pointer and reduce the
double
indirect pointer. This can make the process of accessing the data much faster
because
it will save a 2nd-order scale of access operation.
But we afraid the space is not enough for a super large file so we hold on to
having
a doublly_indirect pointer.

               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 "threads/thread.h" newly added struct:
struct thread
{
    ......
    struct dir *directory   /*record the working directory of current thread*/
    ......
};
```

---- ALGORITHMS ----

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

The function is called get_directory_from_path.
We require a path, an string to store the parsed filename and the dir the filename
belongs to.
We first read in the path, check if it's absolute path, if so, we open the root
otherwise we open the working direct of the process, if it doesn't have one, we open
and set the root as working dir for it.

Then we parse the filepath from the beginning, each time a string before the next '/'
check if there is the required named file or dir in the current dir and if the parsed
filename is a dir name because we will go deeper into dir later. Either case fails will
lead to a NULL return.

With the previous name parsed, we also check if we are at the last dir before the last
dir/file name which is also the required open/create/remove name, if so, we return after
we change dir to this dir and return here.

Before actually return the dir of the desired file/dir, we copy the name required into
the filename string past into the function.

The difference between traversals of absolute and relative paths is whether the parse
start from the root or the working dir of the caller process, as mentioned above.

---- 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.

When we do directory operations we will first acquire an inode lock that
prevent other processes from accessing this inode that directory has.The
lock will be released after all operations is done.

>> 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,once we open or reopen a inode its open_cnt indicater will be increased
by one. We have a function that if the inode's open_cnt is larger than one
(one for directory remove itself) it will return false and if so we won't
let dir_remove remove the directory.

```
---- RATIONALE ----

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

we choose to initiate the current thread's working directory in thread init by
calling dir_open_current and init the initial thread by create a function that
open the root directory when we first init everthing in init.c. And we close
the dir when process is exited. We choose this way since start_process can only
take one viod arguement and we already take the file name for that. It is hard
to keep every threads with right working directory when create a new thread.

                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 "filesys/cache.c" newly defined struct member:

struct cache_line
{
    uint8_t data[BLOCK_SECTOR_SIZE];   /*data in this cache line*/
    block_sector_t sector;             /*the sector that cache line has*/
    int waiters;                       /*number of users that is waiting for
reading/writing from/to this cache line*/
    int indicator;                     /*indicator that shows whether cache line
is exclusive or not*/
    struct lock cache_line_lock;       /*lock to protect data in cache line*/
    struct lock bool_lock;             /*lock to protect all bool indicator
variables*/
    struct condition waiting_queue;    /*readers/writers wating in this
condition variable to avoid racing*/
    bool accessed;                     /*whether the cache line is accessed (for
eviction)*/
    bool dirty;                        /*whether the cache line is modified*/
    bool used;                         /*whether the cache line is used*/
};

/*readhead element*/
struct ahead
{
    block_sector_t sector;  /*read head sector*/
    struct list_elem elem;  /*store it in the list*/
};

In "filesys/cache.c" newly declared global or static variable:

/*create the cache*/
struct cache_line cache[MAX_CACHE];
/*lock on cache prevents from racing when going through*/
struct lock cache_lock;
/*list of sectors to be read ahead*/
static struct list ahead_list;
/*read head list lock*/
static struct lock ahead_lock;
/*indicate whether to read or not*/
static struct condition ready;
/*index indicator used in eviction*/
int index;
```

---- ALGORITHMS ----

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

We use clock algorithm like the one we use in project 3 for frame eviction. When we first get the sector for allocation we will check if this sector is already assigned with a cache line. If not we will check if there are any empty cache line for allocation. If both of these two case are false we will go throught the whole cache twice use clock algorithm for eviction. For first go through we set access bool indicator to false and go through again if one cache line is still with false access indicator we will evict it and if it is dirty we will write it back to the sector it belongs to.

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

A cache line will be write back to the disk only when the filesys_close() is called or cache line needs to be evicted or the guard thread periodicly clear the whole cache.

When a file is close or after a time period, we will go through the whole cache and write all cache line back into the sector it belongs to and wake all process that is waiting and reset all cache line indicator variable.

When a eviction happens if the cache line needs to be evict and it is dirty we will write it back to the sector it belongs to and reassign the cache line.

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

We create a thread as a guard thread that is always running and waiting if the read head list is empty when init cache.When read block is called it will add the
block that is to be read and the next block into the read ahead list. Read ahead guard threadwill be awake now and allocate cache line for block and get its data from the sector.

---- 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?

We have a lock for each cache line. We need to acquire the lock when modfying the cache line. When we call eviction we will first try to acquire the lock if we can't acquire the lock then we will skip this cache line. So there will be no racing when eviction.

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

Since we will first acquire the lock when eviction and only when we finish all evicting operation will the lock be released.

---- RATIONALE ----

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

When we repeatedly access one small file for multiple times or we modify one large file using buffer caching will improve the efficiency by avoiding repeatedly
access disk. Read-ahead will benifit large file modification and wirte-behind

```
will
benifit both repeately small file access and large file modification.

               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?
```