---- TEAM ----

>> Team name.
Team 10

>> Fill in the names, email addresses and contributions of your team members.

Tuan Manh Lai <laituan245@kaist.ac.kr> (Contribution1 = 50%)
Nham Van Le <levn@kaist.ac.kr> (Contribution2 = 50%)

>> Specify how many tokens your team will use.

3 tokens

---- 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.
We modified the on-disk inode structure.

- *is_dir* variable of an inode structure lets us know whether the inode currently represents a directory or a normal file. This variable is important for implementing subdirectories.
- *parent* variable of an inode structure lets us know the parent directory. Again, this variable is important for implementing subdirectories.
- *doubly_indirect* variable of an inode structure lets us know the sector number of the doubly indirect block. This variable is important for implementing indexed files.

```
/* On-disk inode.
   Must be exactly DISK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
          int is_dir;
          disk_sector_t parent;
          disk_sector_t doubly_indirect;
          off_t length;                 /* File size in bytes. */
          unsigned magic;                       /* Magic number. */
          uint32_t unused[123];                 /* Not used. */
  };
```

In order to implement extensible files, we also modified the in-memory inode structure. We added one new variable, which is the semaphore *file_growth_sema*. This semaphore is needed to avoids a race if two processes attempt to extend a file at the same time.

>> A2: What is the maximum size of a file supported by your inode structure? Show your work.
In order to eliminate external fragmentation and to support extensible files, we implemented an index structure with only one double indirect block. Each inode structure contains exactly one pointer that points to a sector of pointers that point to other sectors of pointers that then point to sectors of the file's data.

In Pintos, the size of one disk sector is 512 bytes. And so the maximum size of a file supported by our inode structure is *(512 / 4) * (512 / 4) * 512 bytes = 8 MB*. This is sufficient since no file system will be larger than 8 MB and the Pintos manual says that we only need to support files as large as the partition (minus metadata).

Note that in many UNIX-based file systems, inode structure usually has many more pointers for indexing purpose. For example, usually, there are 12 pointers point to direct blocks and 3 pointers point to indirect blocks. However, in this project, we use only 1 pointer because it is sufficient and it makes our code not too complicated.

---- SYNCHRONIZATION ----

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

For each open file, we have a special semaphore associated with it (*file_growth_sema* in struct inode). Whenever a process wants to write some data to a file, it first needs to down the semaphore *file_growth_sema* associated with the file. After that, the process checks if it will write at a position past EOF. If yes, the code for doing file growth will be executed. Otherwise, file growth will not occur. After that, the process will up the semaphore and the actual data writing will actually begin. In this way, the race condition can be avoided. No two processes will actually extend a 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.

>> 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.

---- 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?

Based on the discussion in A2, it is obvious that our inode structure is indeed a multilevel index. Recall that we implemented an index structure with only one double indirect block.
The advantages of this approach are:

+ It is simple. We have only one pointer in our inode structure for indexing purpose.

+ It is sufficient. This Pintos project requires us to support 8 MB files. As discussed in A2, our index structure can indeed support 8MB files.

The disadvantages are:

+ Whether a file is large or not, in order to access the file content for the first time, we always need to do at least 4 disk accesses (one for the inode, one for the doubly indirect block, one for the singly indirect block, and one for data block). We actually don't need that many disk accesses for accessing small files if our inode structure has more pointers for indexing purpose.
+ This approach is not practical. In real systems, there are usually many files that are larger 8MB. This approach is only applicable to this particular Pintos project.

== 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.

We added a variable *cur_dir* to the struct thread to keep track of the current working directory of a process. In addition, some changes needed to implement subdirectories were also mentioned in A1.

---- ALGORITHMS ----

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

In order to traverse a user-specified path, we need some starting point.

+ If the path starts with character '/', then the path is an absolute path (e.g., /a/b/data.txt). In this case, the starting point is the root directory. We can get the root directory by using the function *dir_open_root()*.
+ Otherwise, if the path does not start with the character '/', then the path is a relative path (e.g., a/data.txt). Here, the starting point is the current directory of the process.

Except for the code for determining the starting point, the rest of our code for traversing a general path is the same for both cases. First, we have a pointer *cur* that points to the "starting point" directory. We make use of the function *strtok_r()* to break down the path string into smaller pieces(e.g., /a/b/data.txt will be broken down into 'a', 'b', and 'data.txt'). We just simply iterate over the pieces. If the current token is '..', *cur* will be updated to point to the parent directory. If the current token is '.', *cur* will still be the current

directory. Otherwise, the token should be the name of some file or directory, and so we first try to see if the directory represented by *cur* has an entry for that name (using the function *dir_lookup()*). If no, then we immediately stop traversing the path and do something else. If yes and the name belongs to a file, it must be the last token in path. If yes and the name belongs to a directory, we can simply update *cur*.

---- 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.

>> 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 allow deletion of a directory that is open by a process or in use as a process's current working directory. We modified some functions so that attempts to open files (including '.' and '..') or create new files in a deleted directory must be disallowed.

---- RATIONALE ----

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

We used a struct dir * variable to represent the current directory of a process (*cur_dir* in the struct thread). We chose this approach because it is quite natural and it was the only approach we could think of.

== 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.

We implemented the buffer cache in the files filesys/cache.h and filesys/cache.c.

In filesys/cache.h:

struct cached_sector {
  struct list_elem elem;
  disk_sector_t sector_idx;
  void * data;
  bool dirty;
  struct semaphore sema;
};
	Each instance of this struct represents a cached sector. Note that the variable *dirty* keeps track of whether the cached sector is dirty or not.

In filesys/cache.c:

static struct lock cache_lock;	A lock needed to prevent some race conditions that involve the buffer cache.
static struct list sectors_list;	A list of all cached sectors in the buffer cache. Conceptually, this list is our buffer cache.
static int count;	The number of cached sectors in our buffer cache (This number should not be larger than 64).

---- ALGORITHMS ----

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

Because the buffer cache is limited to 64 sectors in size, there are situations where we need to evict some cached block in order to be able to fetch a new block from disk. We implemented a FIFO replacement algorithm. We maintain a list of cached sectors (*sectors_list* defined in cache.c). When we want to fetch a new block from disk, we will first check whether the cache is full (i.e., the list has 64 elements). There are two cases:

+ If the cache is not full, then we can simply fetch the new block and insert it to the end of the list.
+ If the cache is full, then we evict the first block at the beginning of the list. After that, we can fetch the new block and insert it to the end of the list.

In this way, new blocks are always near the end of the list and old blocks are near the beginning of the list. And the list behaves just like a FIFO queue.

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

The basic idea of write-behind is that we should keep dirty blocks in the cache, instead of immediately writing modified data to disk. We only write dirty blocks to disk in three cases:

.

1. When a cached sector is about to be evicted, we need to write the content of the sector to disk if it is dirty. As mentioned in C1, for each cached sector, we have a variable that keeps track of

whether the cached sector is dirty or not. For writing data to disk, we simply use the function *disk_write()*.

2. We need to periodically write all dirty, cached blocks back to disk. Before kernel initialization is about to finish, we initialize the buffer cache and also start a new thread whose main job is to periodically write all dirty, cached blocks back to disk. This is illustrated by the code listing below.

```
void buffer_cache_init() {
  ...
  thread_create("periodical_write_behind", PRI_DEFAULT, flush_periodically, NULL);
}

static void flush_periodically(){
  while(true) {
          timer_sleep(10); // Make use of function timer_sleep() implemented in project 1
          flush();              // Write all dirty, cached blocks back to disk
  }
}

void flush() {
  lock_acquire(&cache_lock);
  struct list_elem *e;
  for (e = list_begin(&sectors_list); e != list_end (&sectors_list); e = list_next (e)) {
          struct cached_sector * tmp = list_entry(e, struct cached_sector, elem);
          sema_down(&tmp->sema);
          if (tmp->dirty) {
            disk_write(filesys_disk, tmp->sector_idx, tmp->data);
            tmp->dirty = false;
          }
          sema_up(&tmp->sema);
  }
  lock_release(&cache_lock);
}
```

3. The cache should also be written back to disk in filesys_done() (i.e., halting Pintos flushes the cache). This is trivial. We just need to look at every cached sector and see if it dirty or not. If it is dirty, we write its data to disk by using the function *disk_write()*. We actually also make use of the function *flush()* here.

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

We did not implement read-ahead because we did not have enough time and our code still passed all the tests even without the read-ahead functionality. However, probably, one way to implement read-ahead could be to start a new thread when one block of a file is read. The job of this new thread is to fetch the next block of the file into the cache. Note that this approach is somewhat similar to our approach for implementing write-behind (discussed in C3).

---- 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?
For each cached sector, we have a special semaphore called *sema* associated with it.

Before a process can start reading or writing data in the cached sector, the process first needs to down the semaphore *sema* of the cached sector. Once the cached sector is done being used by the process, the process will up the semaphore.

In addition, when some process wants to evict some cached sector, the process also first needs to down the semaphore *sema* of the cached sector.

In this way, when one process is actively reading or writing data in a buffer cache sector, the other process that wants to evict that sector will need to wait before the cached sector is done being used. In other words, it will need to wait until the semaphore *sema* of the cached sector is upped.

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

The answer to this question is quite similar to C5. By having a special semaphore called *sema* for each cached sector, we can prevent processes from attempting to access a cached sector that is being evicted.

---- RATIONALE ----

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

A file workload that likely to benefit from buffer caching would be where some particular data blocks in some files needs to be read/written very frequently. A file workload that likely to benefit from read-ahead is one that reads a file sequentially. A file workload that likely to benefit from write-behind would be where some particular data blocks in some files needs to be written very frequently.