| PROJECT 4: FILE SYSTEMS |

---- GROUP 12----

Sakaar Khurana 10627
Nikhil Aggarwal 10446
Aditi Krishn 10039

INDEXED AND EXTENSIBLE FILES

---- DATA STRUCTURES ----

1. in inode.c

1.1 #define INODE_BLOCKS 124

This is the number of direct blocks in an on disk inode

1.2
int INODE_LIST = BLOCK_SECTOR_SIZE / sizeof(block_sector_t);

This is the number of blocks of the indirect inode blocks

1.3
```
struct inode_disk
  {
    unsigned number_blocks;
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    block_sector_t sector[INODE_BLOCKS]; /* for the block numbers */
    enum file_type type;
  };
```

The new inode_disk structure which contains simple information about the file it represents, length, numberofblocks it occupies,
a sector array for direct, 1 indirect, and 1 double indirect block.

1.4
```
/* On-disk inode list for linked and or double linked lists.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_list
  {
    block_sector_t sector[128];         /* for the block numbers */
  };
```

This represents the linked or double linked block that I store on the disk.

1.5 in struct inode:
```
//struct inode_disk data;             /* Inode content. */
 struct lock inode_lock;
```

We removed the reference to inode disk in the inode because it will be accessed over the cache and the sector in the inode.
We added a lock to the structure for synchronization

The maximum size of a file is:

A) direct blocks 124
B) single indirect blocks 128
C) double indirect blocks 128 * 128

=> 124 + 128 + 128 * 128 = 16636 Blocks which is
   16636 * 512 = 8.517.632 bytes or 8.318 MB

---- Multi level indexing ----

We support multilevel index up to one doubly indirect block.
It is the standard way in linux and easy to manage.
The blocks can be allocated and extended in a consistent way.

---- TEST CASES ----
- All the files when are brought into the filesystem call the function
  file_grow in filesys/inode.c where data is allocated as per need.
  eg. run following from userprog/build:
      pintos -p ../../examples/echo -a echo -- -q
- Files can also be rown after cretion time usig same general function.

SUBDIRECTORIES

---- DATA STRUCTURES ----

1. in inode.h

```
enum file_type
{
      FILE_FILE,
      FILE_DIR
};
```

This enumeration tells if an inode is a directory or file

2. in thread.h, structure thread:

```
struct dir * pwd;
```

The current working directory of the thread.

---- ALGORITHMS ----

- First of all we call a parser function to remove any multiple entries
  of / in the filename. (in filesys.c, parse_filename()).
- Then we call the function dir_lookup_rec() in directory.c which will
  does the traversing.
- It will look for a / in the first character of the filename, if it
exists
  we know that we have an absolute path and start at root otherwise use
  the current working directory as starting point.
- We strtok the filename and walk through the directories in the tokens
one
  by one to see if they are a directory.

---- Explaining the Directory Structure ----

- We are saving a struct dir in the thread structure that represent the current working directory.
- This way it is easy for us to use it when a user opens a file or directory with relative pathname.
- We just grab the directory and go from there.

---- TEST CASES ----
- mkdir.c: makes a directory passed to it.
- chdir.c: changes the directory to the directory passed to it.
- open.c: changes to a directory 'dir', makes 2 directories over there, changes directory back to the root directory, opens a file with directory 'dir' and calls readdir over there.
  This test case tests the functionality of all the system-calls.

BUFFER CACHE

---- DATA STRUCTURES ----

1. in filesys/cache.c

1.1
```
struct cache_block {
  //block sector on disk
  block_sector_t bid;
  //corresponding kernel page
  void * kpage;
  //fields to check access and if someone wrote to the page
  bool dirty;
  bool accessed;
  int reader;
  int writer;

};
```

This represents an entry in the cache table.
It has the corresponding block number on the disk, a kpage for the kernel page where it is stored right now.
A dirty and accessed variable for writing and eviction and reader and writer are used for synchronization.

1.2
```
//how many blocks is one page
int SECPP;
```

The global variable tells the number of blocks that fit in one kernel page.

1.3
```
//cache array
struct cache_block * cache[cache_size];
```

This is the actual cache table.

1.4
```
//bitmap to identify free entries
struct bitmap * cache_table;
```

The bitmap that is used to identify empty entries in the cache table.

1.5
struct lock cache_globallock;

A global lock to protect the cache table when I add new elements or

2. in filesys/inode.h

2.1
//for the read ahead thread
struct lock lock_readahead;

A lock for the read ahead thread.

2.2
struct list list_readahead;

A list for the read ahead thread.

2.3
struct condition cond_readahead;

A condition variable to signal the waiting read ahead thread to wake up.

2.4
//structure for the readahead list
struct readahead {

```
block_sector_t bid;
struct list_elem elem;
```

};

An element in the read ahead list.

---- ALGORITHMS ----

- Eviction Algorithm:
  - We use the second chance algorithm from project 3. Every block has an accessed variable. A prerequisite for a page in order to get evicted is that the page has no reader or writer at the moment.
  - If the accessed bit is true We reset it to fault and leave the block in the list.
  - If one block has a false accessed bit we evict it.

---- Working of Buffer cache ----

- When a process needs to access the same part of a file multiple times or
  when multiple processes access the same part of a file a lot then caching
  will be a lot faster then reading or writing to the disk directly.
- Also when a process changes a lot of bytes in the same block a cache will
  be useful to prevent slow reads and writes form the disk.

- Read ahead will have its benefits when a process needs to read contiguous
  blocks from a file, so from 1 to last byte.

---- TEST CASES ----
Interation betweeen kernel and filesystem occurs through buffer cache, so all user programs running properly signifies correct working of buffer cache.