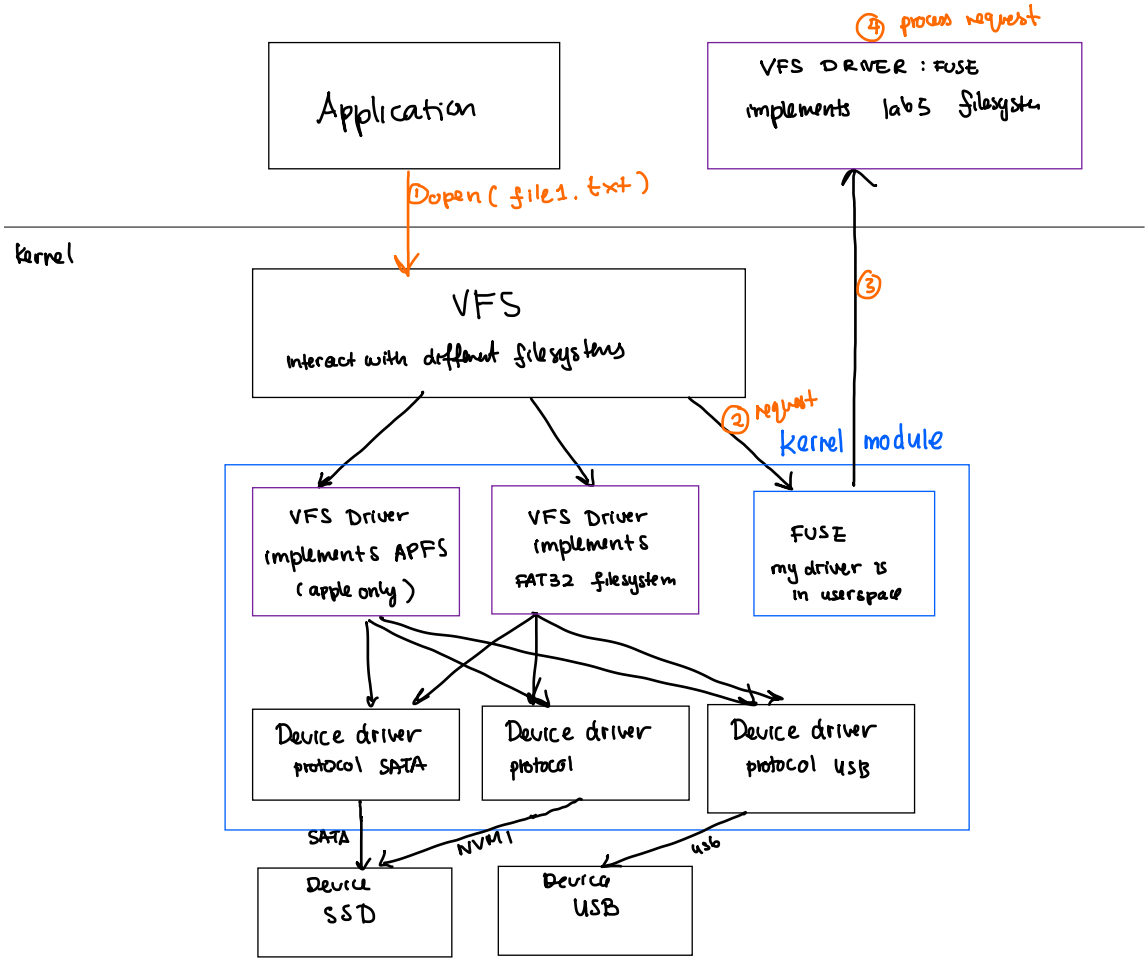
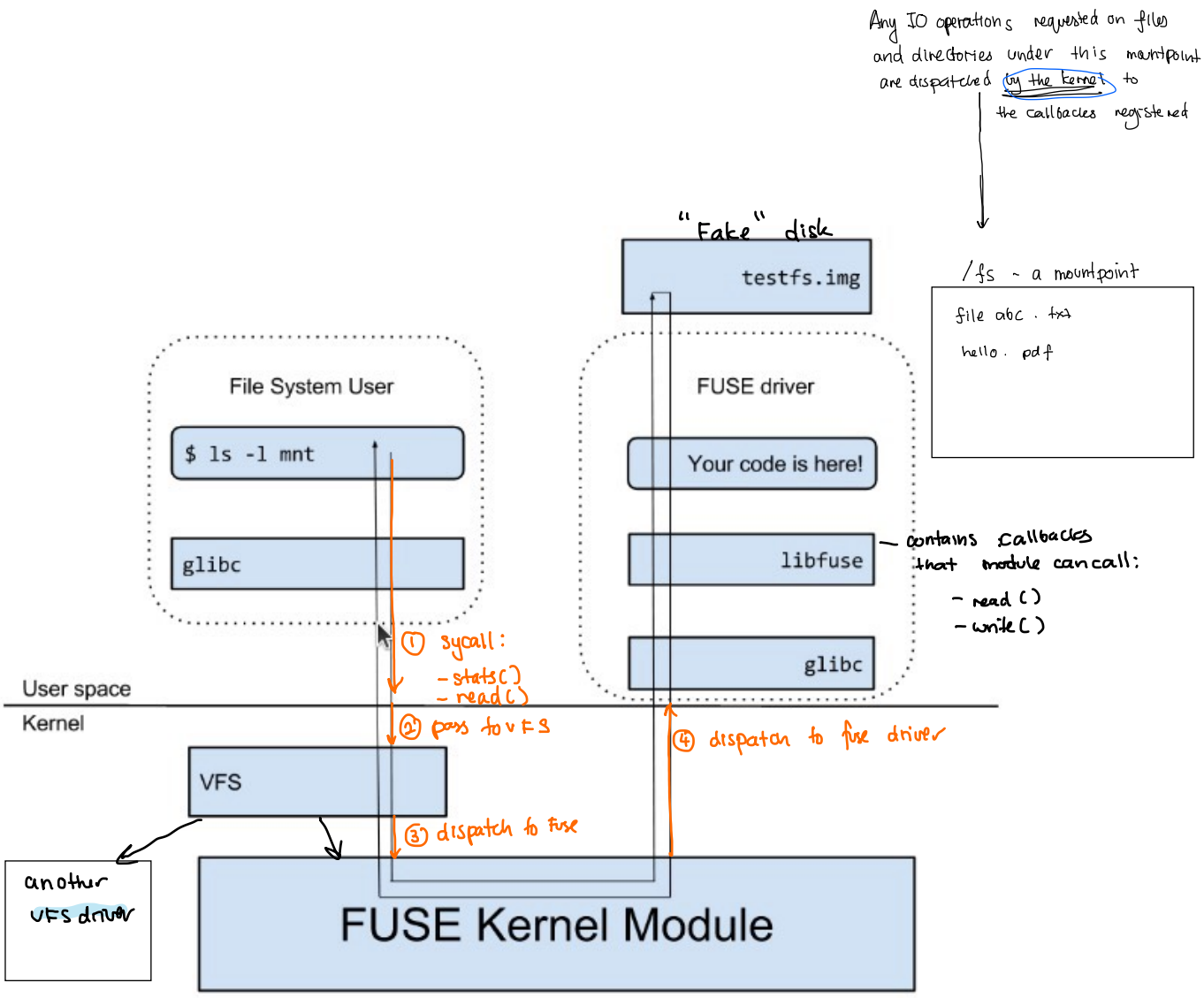
Lab5 Fuse



FUSE allows developers to create and implement file systems without having to write kernel-level code. This approach can simplify development and testing while improving system stability, as errors in the user-space FUSE file system driver are less likely to cause kernel panics or crashes compared to a kernel-level file system implementation.

1. User applications perform file system operations.
2. The VFS layer in the kernel handles the operations and communicates with the appropriate VFS driver. In this case, it's the FUSE kernel module.
3. The FUSE kernel module forwards the requests to the FUSE file system driver in user space.
4. The FUSE file system driver processes the requests and performs the necessary operations.


Any IO operations requested on files and directories under this mountpoint are dispatched by the kernel to the callbacks registered
"Fake" disk
testfs.img
/fs - a mountpoint
file abc.txt
hello.pdf
File System User
$ ls -l mnt
glibc
FUSE driver
Your code is here!
libfuse
contains callbacks that module can call:
- read()
- write()
glibc
① syscall:
- stats()
- read()
User space
Kernel
② pass to VFS
④ dispatch to fuse driver
VFS
③ dispatch to fuse
another VFS driver
FUSE Kernel Module

```
$ mkdir fs     // create a directory to serve as a mount point

$ df fs        // see what file system fs is associated with
Filesystem     1K-blocks    Used Available Use% Mounted on
/dev/sda1        7092728 4536616   2172780  68% /

$ ls fs        // notice, "fs" is empty

① $ ./lab5fuse -image test.img fs    // mount test.img at fs

② $ df fs      // this ends up with an error because you haven't implemented
               // but notice that fs's file system is different.
df: fs: Operation not supported

$ ls fs        // similarly, "ls" reports an error as well
ls: cannot access 'fs': Operation not supported

$ ls -l        // you will see that fs is in an unknown state
...
d????????? ? ?      ?            ?            ? fs
...

$ fusermount -u fs  // now unmount fs

$ ls fs        // and "fs" is an empty dir again
```

the Kernel will dispatch
- readdir()
- statfs()

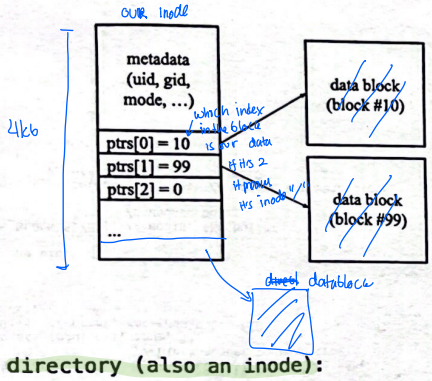
make to fuse driver

calls that "ls fs"

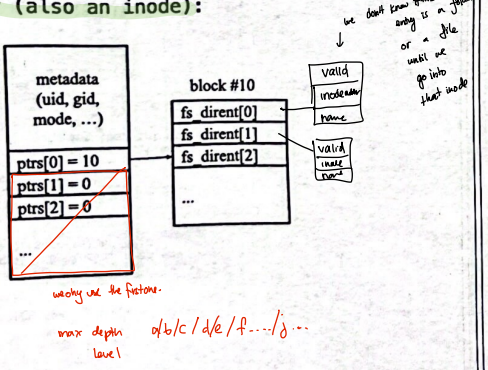
- but we haven't implemented

1. fs5600 visualization

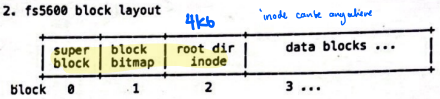
fs5600 file inode:



fs5600 directory (also an inode):

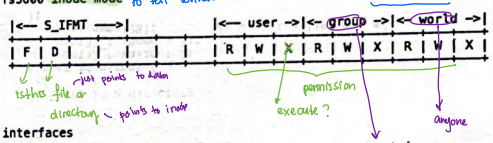


2. fs5600 block layout



3. fs5600 inode mode

to tell which block has been used



4. interfaces

```
fs_init - constructor (i.e. put your init code here)
fs_statfs - report file system statistics
fs_getattr - get attributes of a file/directory
fs_readdir - enumerate entries in a directory
fs_read - read data from a file
fs_rename - rename a file
fs_chmod - change file permissions

fs_create - create a new (empty) file
fs_mkdir - create new (empty) directory
fs_unlink - remove a file
fs_rmdir - remove a directory
fs_write - write to a file
fs_truncate - delete the contents of a file
fs_utime - change access and modification times
```

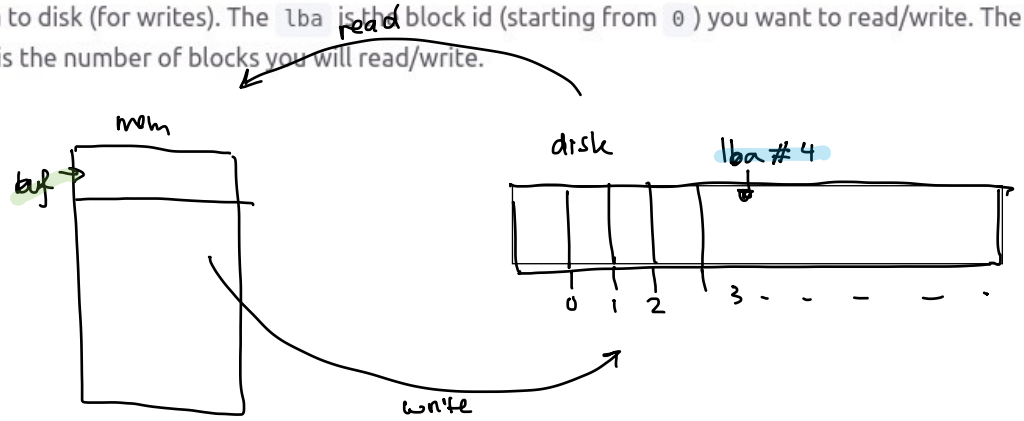
Content Overview

fs5600 disk interface

The fs5600 disk is abstracted as an array of blocks. The size of each block is `4KB` (4096 bytes or `FS_BLOCK_SIZE`). You can read/write disk by two simple interfaces:

```
int block_read(void *buf, int lba, int nblks);
int block_write(void *buf, int lba, int nblks);
```

The `buf` is a pointer to memory where you want to store the data from disk (for reads) or contains the data to disk (for writes). The `lba` is the block id (starting from `0`) you want to read/write. The `nblks` is the number of blocks you will read/write.



File system format

The format of fs5600 is as follows:

- the first block (block 0) is the superblock;
- the second block (block 1) is the bitmap block;
- the third block (block 2) is the root directory;
- other blocks are blocks that contain either inodes or data.

```
   +-------+--------+----------+------------------------+
   | super | block  | root dir |     data blocks ...    |
   | block | bitmap |   inode  |                        |
   +-------+--------+----------+------------------------+
block     0        1          2          3 ...
```

Superblock and block bitmap

Superblock

The superblock is the first block in the file system, and contains the information needed to find the rest of the file system structures. The following C structure (see `fs5600.h`) defines the superblock:

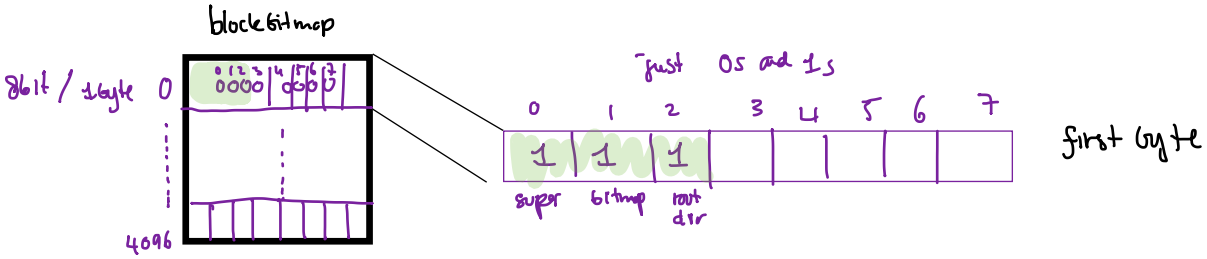
```
struct fsx_superblock {
    uint32_t magic;            /* 0x30303635 - some magic number */
    uint32_t disk_size;        /* in 4096-byte blocks */
    char pad[4088];            /* to make size = 4096 */
};
```

Note that `uint32_t` is a standard C type found in the `<stdint.h>` header file, and refers to an unsigned 32-bit integer. (similarly, `uint16_t`, `int16_t` and `int32_t` are unsigned/signed 16-bit ints and signed 32-bit ints)

Block bitmap

This block bitmap indicates which blocks have been used and which are free. One bit represents one block: the first bit represents the first block (block 0), the second bit represents the second block (block 1), and so forth. If a bit is `0`, the corresponding block is free; otherwise, the block is used.

In fs5600, the first three bits are always `1` because they are (1) the super block, (2) the block bitmap itself, and (3) the root dir inode. (the bits for blocks 0, 1 and 2 will be set to 1 when the file system is created, so you don't have to worry about excluding them when you search for a free block.)



3.2 Block allocation

Block allocation (with block bitmap)

To allocate blocks from disk, you need to manipulate block bitmap.

The bitmap is cached in a 4KB memory buffer (a global variable `block_bitmap`). You will need keep the cache in sync by writing it back to disk after each block allocation and free.

You're given the following functions to handle the bitmap in memory:

block_bitmap
which bit to test / set or clear

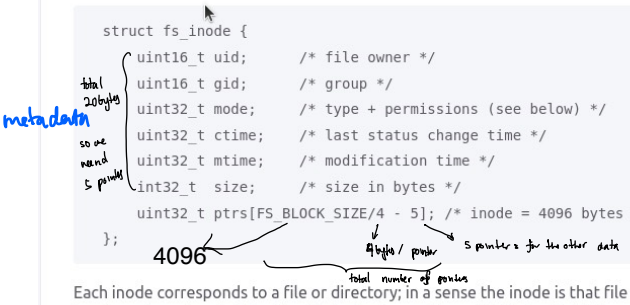
```
int bit_test((void*)map, int i);
void bit_set(unsigned char *map, int i);
void bit_clear(unsigned char *map, int i);
```

A side note: using a single block for the bitmap means that the maximum number of disk blocks fs5600 can track is `32768 (= 4096 * 8)`. Each block is 4KB. Hence the maximum size of an fs5600 is `128MB (= 32768 * 4KB)`.

3.3 Inode

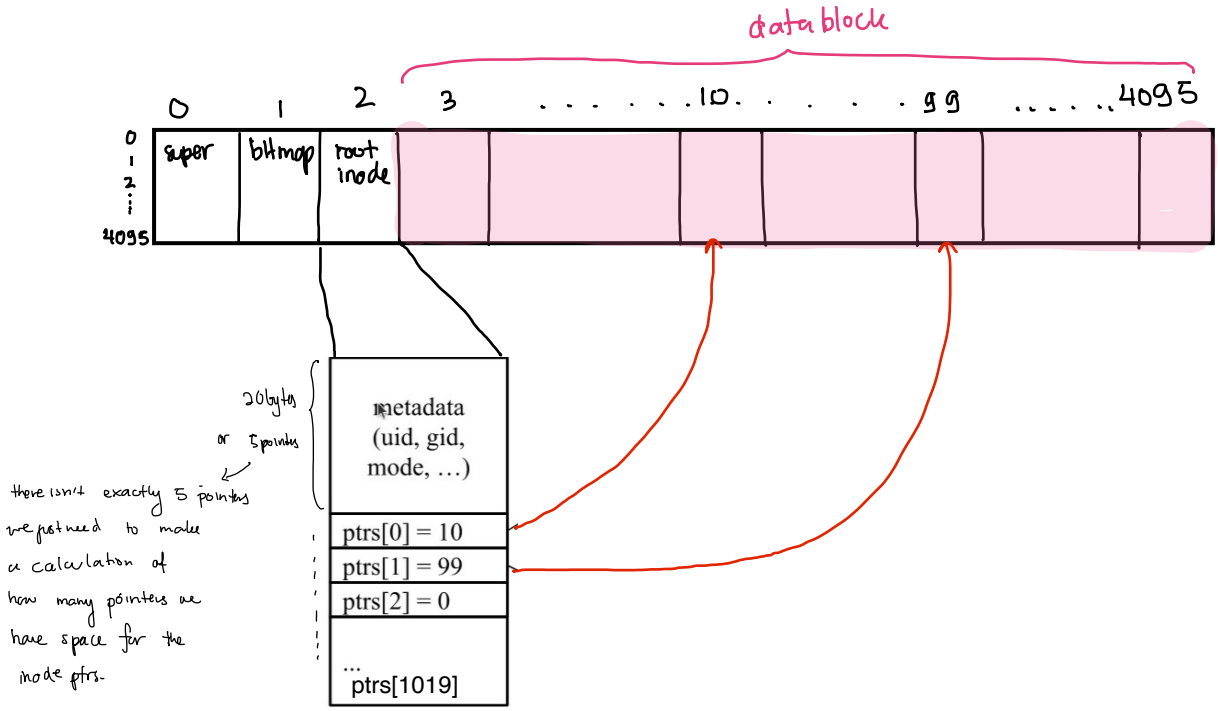
In particular, inode is defined as follows (see also `fs5600.h`):

```
struct fs_inode {
    uint16_t uid;      /* file owner */
    uint16_t gid;      /* group */
    uint32_t mode;     /* type + permissions (see below) */
    uint32_t ctime;    /* last status change time */
    uint32_t mtime;    /* modification time */
    int32_t  size;     /* size in bytes */
    uint32_t ptrs[FS_BLOCK_SIZE/4 - 5]; /* inode = 4096 bytes */
};
```



Each inode corresponds to a file or directory; in a sense the inode is that file or directory. In fs5600, inode can be uniquely identified by its inode number, which is its block id. Again, **fs5600 uses block ids as inode numbers.** The root directory is always found in inode `2`; inode `0` (the super block) is invalid and can be used as a `NULL` value.

Hint: a path lookup always begins with inode `2`, the root directory.



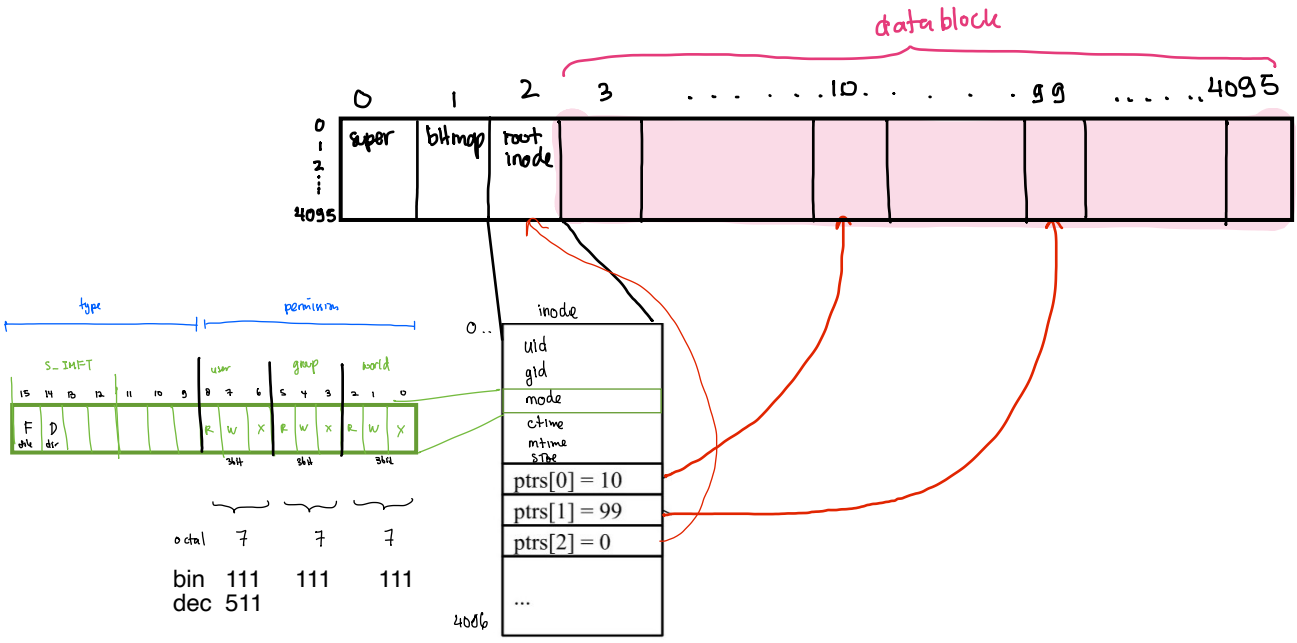
indoe’s mode

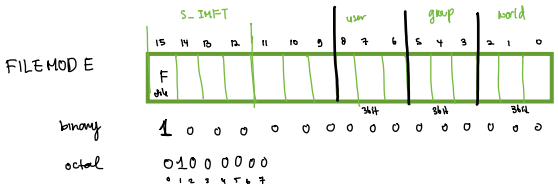
File mode

The FUSE APIs (and Linux internals in general) mash together the concept of object type (file/directory/device/symlink…) and permissions. The result is called the `file mode` (the attribute `mode` in the `fs_inode` above), and looks like this:

```
|<-- S_IFMT --->|          |<-- user ->|<- group ->|<- world ->|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| F | D |   |   |   |   |   | R | W | X | R | W | X | R | W | X |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

```
00000000 00000000 10000011 10000000
```





Binary: 1000 0000 0000 0000

Grouped: 001 000 000 000 000 000

Now convert each group of three binary digits to its octal equivalent:

Octal: 1 0 0 0 0 0

Since it has multiple 3-bit fields, it is commonly displayed in base 8 (octal). For example, permissions allowing `RWX` for everyone (`rwxrwxrwx`) are encoded as `777` (an octal number). Note that, in C, octal numbers are indicated by a leading `0` , e.g., `0777` , so the expression `0777 == 511` is true.

The `F` and `D` bits correspond to `0100000` and `040000` , indicating if the inode is a regular file (`F` bit is set) or a directory (`D` bit is set).

> *A side note*: since the demise of 36-bit computers in the early 70s, Unix permissions are about the only thing in the world that octal is used for.

There are a bunch of macros (in `<sys/stat.h>`) you can use for looking at the mode; you can see the official documentation with the command `man inode` , but the important ones are:

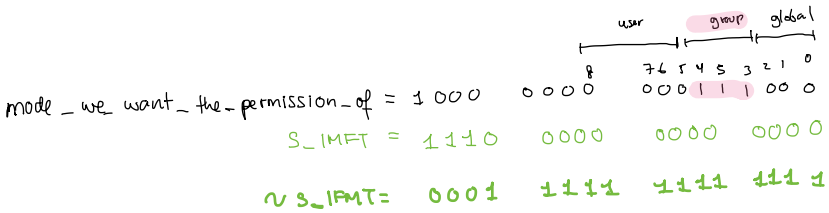
- `S_ISREG(m)` - is it (i.e. the inode with mode `m`) a regular file?
- `S_ISDIR(m)` - is it a directory?
- `S_IFMT` - bitmap for inode type bits.
  You can get just the inode type with the expression `m & S_IFMT` , and just the permission bits with the expression `m & ~S_IFMT` . (note that `~` is bitwise NOT - i.e. `0` s become `1` s and `1` s become `0` s)

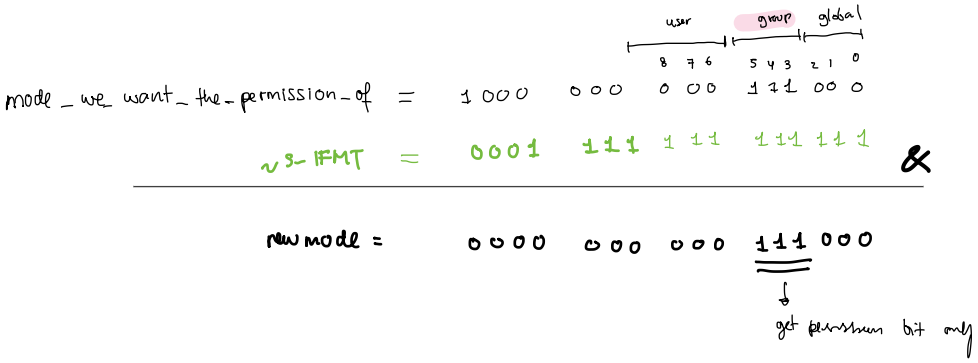
① what is the type of this mode

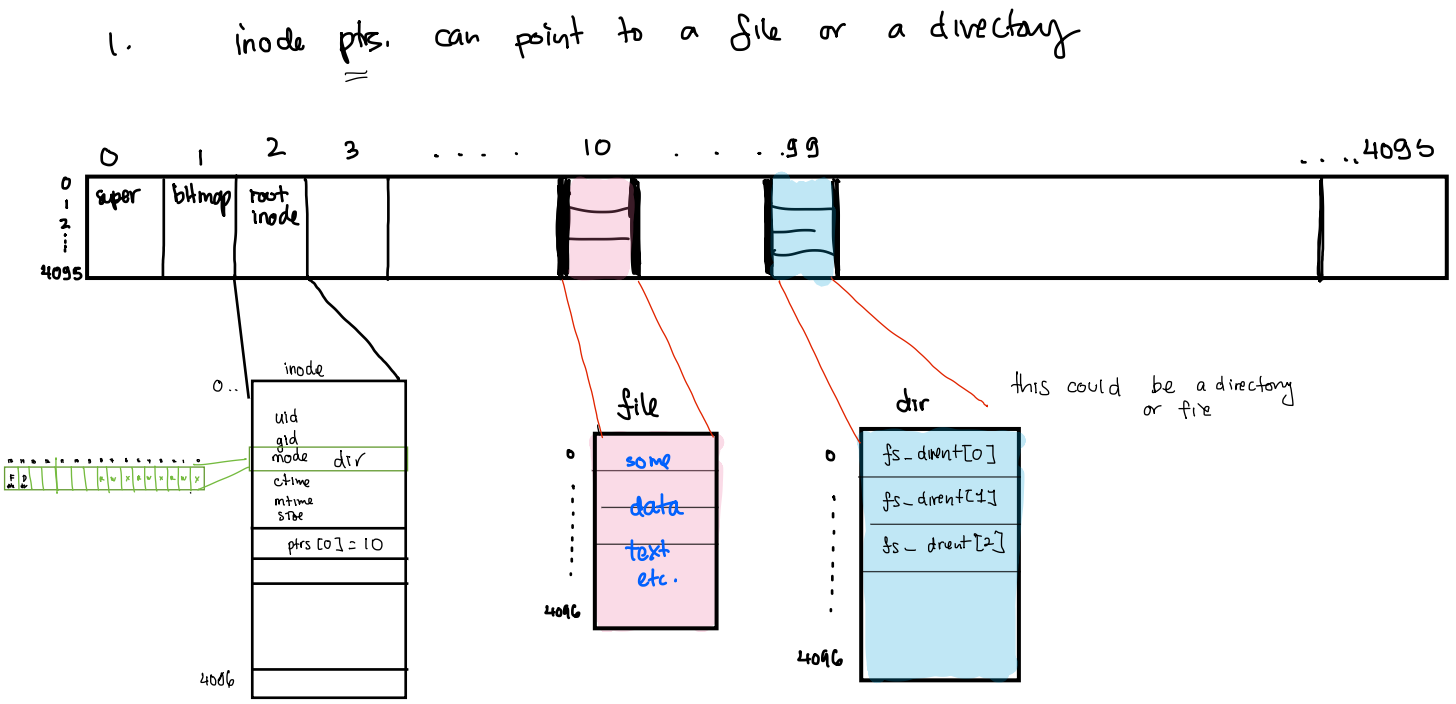
↓

inode type = m & S_IFMT

② set the permission bits

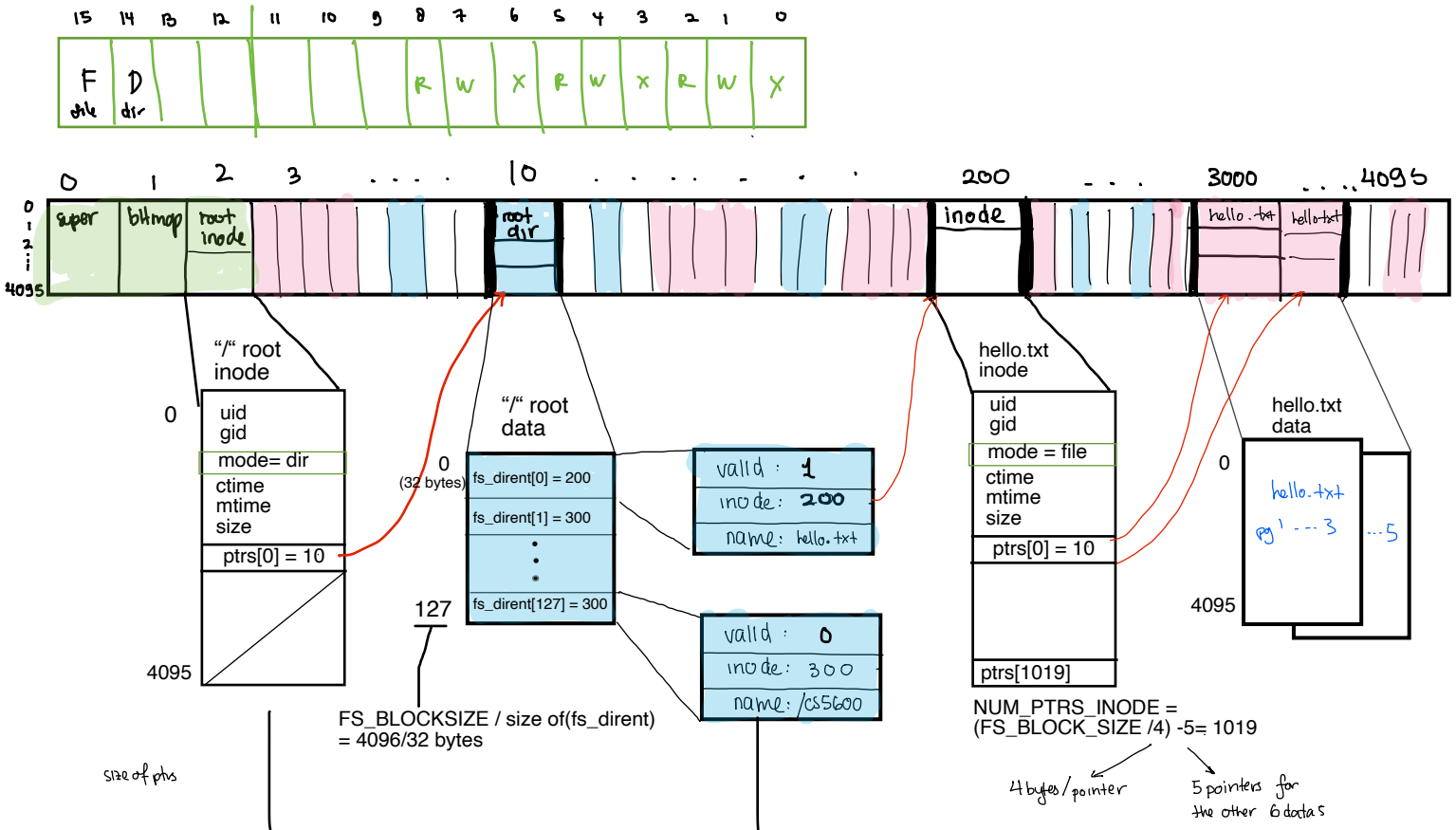






Each directory entry maps a human-readable name (a string) to an inode number (again, an inode number is a block id). It is defined as follows (see also `fs5600.h`):

```
struct fs_dirent {
    uint32_t valid : 1;  /* the first bit of uin32_t */
    uint32_t inode : 31; /* the next 31 bits of uint32_t */
    char name[28];       /* with trailing NULL */
};
```




```
struct fs_inode {
    uint16_t uid;
    uint16_t gid;
    uint32_t mode;
    uint32_t ctime;      // time of l
    uint32_t mtime;      // last modi
    int32_t  size;
    uint32_t ptrs[NUM_PTRS_INODE]; /
};
```

```
52  struct fs_dirent {
53      uint32_t valid : 1;
54      uint32_t inode : 31;
55      char name[28];
56  };
```

1bit

31 bits can hold 2^31 addresses of datablock

28 byte

Each `fs_dirent` is `32 bytes`, giving `128 (=4096/32)` directory entries in one dir. Why? This is a design choice of fs5600: **fs5600 directories always have only one data block,** meaning only the first pointer (`ptrs[0]`) in a dir's inode contains a valid block id; others are invalid (`ptrs[i] == 0` where `i>0`). That said, fs5600's dir has one data block, which is `4KB`, and one direntry is `32B`, hence a dir only supports a maximum of 128 files.

A directory contains `(name, inode)` pairs, which capture by `fs_dirent`. In particular,

- the `valid` bit indicates if directory entries are being used (`1` means yes; `0` means no).
- the `inode` contains a block id on disk that is where the file's inode locates. Note: in fs5600, an inode is a block (4KB in size) and a inode number is the block id on disk.
- `name` is the file name. The maximum name length is `27 bytes`, allowing entries to always have a terminating `\0` so you can use string functions (e.g., `strcmp`, `strdup`) without any complications.

EX1.1 Init the file system

```c
/* init - this is called once by the FUSE framework at startup.
 *
 * The function reads the superblock and check if the magic number matches
 * FS_MAGIC. It also initializes two global variables:
 * "num_blocks" and "block_bitmap".
 *
 * notes:
 *   - use "block_read" to read data (if you don't know how it works, read its
 *     implementation in misc.c)
 *   - if there is an error, exit(1)
 */
void* fs_init(struct fuse_conn_info *conn)
{

    struct fs_super sb;
    // read super block from disk
    if (block_read(&sb, 0, 1) < 0) { exit(1); }

    // check if the magic number matches fs5600
    if (sb.magic != FS_MAGIC) { exit(1); }

    // EXERCISE 1:
    //  - get number of block and save it in global variable "numb_blocks"
    //    (where to know the block number? check fs_super in fs5600.h)
    num_blocks = sb.disk_size; // from superbloc

    //  - read block bitmap to global variable "block_bitmap"
    //    (this is a cache in memory; in later exercises, you will need to
    //    write it back to disk. When? whenever bitmap gets updated.)
    if (block_read(block_bitmap, 1, 1) < 0) {exit(1);}

    return NULL;
}
```

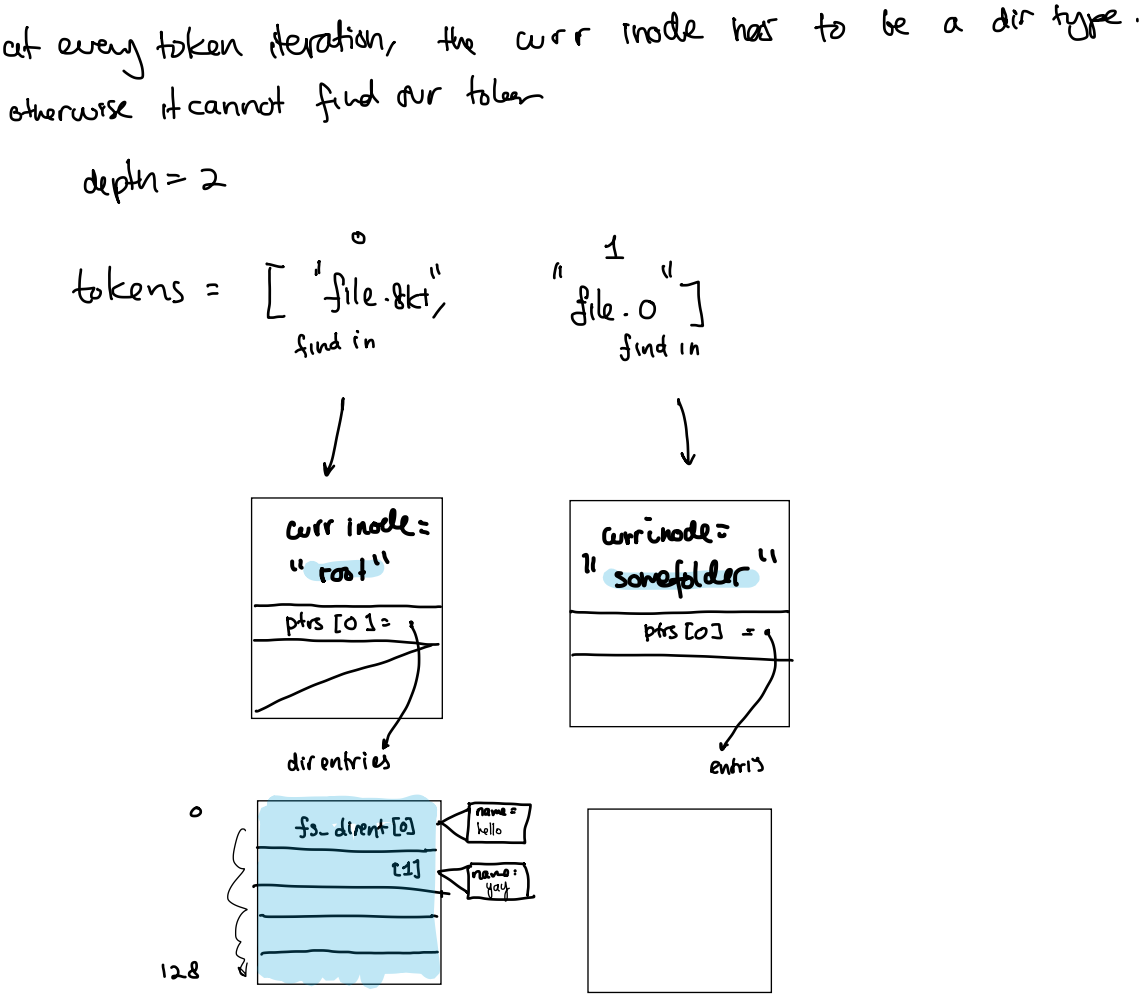
EX1.2 Fill in the file system statistics

```c
/* EXERCISE 1:
 * statfs - get file system statistics
 * see 'man 2 statfs' for description of 'struct statvfs'.
 * Errors - none. Needs to work.
 */
int fs_statfs(const char *path, struct statvfs *st)
{
    /* needs to return the following fields (ignore others):
     *   [DONE] f_bsize = FS_BLOCK_SIZE
     *   [DONE] f_namemax = <whatever your max namelength is>
     *   [TODO] f_blocks = total image - (superblock + block map)
     * # total data blocks in fs
     *   [TODO] f_bfree = f_blocks - blocks used
     *   [TODO] f_bavail = f_bfree
     *
     * it's okay to calculate this dynamically on the rare occasions
     * when this function is called.
     */

    st->f_bsize = FS_BLOCK_SIZE;
    st->f_namemax = 27;  // why? see fs5600.h
    int used_blocks = 0;
    // for each block
    for (int i = 0; i < num_blocks; i++) {
        // iterate through each bit
        for (int j = 0; j < 8; j++) {
            if (block_bitmap[i] & (1 << j)) {
                used_blocks++;
            }
        }
    }
    st->f_blocks = num_blocks;
    st->f_bfree = st->f_blocks - used_blocks;
    st->f_bavail = st->f_bfree;
    return 0;
}
```

```c
struct statfs {
    __fsword_t f_type;    /* Type of filesystem (see below) */
    __fsword_t f_bsize;   /* Optimal transfer block size */
    fsblkcnt_t f_blocks;  /* Total data blocks in filesystem */
    fsblkcnt_t f_bfree;   /* Free blocks in filesystem */
    fsblkcnt_t f_bavail;  /* Free blocks available to
                             unprivileged user */
    fsfilcnt_t f_files;   /* Total inodes in filesystem */
    fsfilcnt_t f_ffree;   /* Free inodes in filesystem */
    fsid_t     f_fsid;    /* Filesystem ID */
    __fsword_t f_namelen; /* Maximum length of filenames */
    __fsword_t f_frsize;  /* Fragment size (since Linux 2.6) */
    __fsword_t f_flags;   /* Mount flags of filesystem
                             (since Linux 2.6.36) */
    __fsword_t f_spare[xxx];
                          /* Padding bytes reserved for future use */
};
```

Ex 2:1

Part 1 helper: path2inum(path):



test1.c:223:F:read_mostly:path_errs:0: Assertion 'rv == -bad1[i].err' failed: rv == -2, -bad1[i].err == -20

/file.8k+/file.0 shuld give -20 ENOTDIR
but mins shows ENOENT

Test cases:

path uid gid mode size ctime mtime
"/", 0, 0, 040777, 4096, 1565283152, 1565283167

"/file.1k", 500, 500, 0100666, 1000, 1565283152, 1565283152

"/file.10", 500, 500, 0100666, 10, 1565283152, 1565283167

"/dir-with-long-name", 0, 0, 040777, 4096, 1565283152, 1565283167

"/dir-with-long-name/file.12k+", 0, 500, 0100666, 12289, 1565283152, 1565283167

"/dir2", 500, 500, 040777, 8192, 1565283152, 1565283167

"/dir2/twenty-seven-byte-file-name", 500, 500, 0100666, 1000, 1565283152, 1565283167

"/dir2/file.4k+", 500, 500, 0100777, 4098, 1565283152, 1565283167

"/dir3", 0, 500, 040777, 4096, 1565283152, 1565283167

"/dir3/subdir", 0, 500, 040777, 4096, 1565283152, 1565283167

"/dir3/subdir/file.4k-", 500, 500, 0100666, 4095, 1565283152, 1565283167

"/dir3/subdir/file.8k-", 500, 500, 0100666, 8190, 1565283152, 1565283167

"/dir3/subdir/file.12k", 500, 500, 0100666, 12288, 1565283152, 1565283167

"/dir3/file.12k-", 0, 500, 0100777, 12287, 1565283152, 1565283167

"/file.8k+", 500, 500, 0100666, 8195, 1565283152, 1565283167

helper we can use:

```
Because tests of the above form are common, additional  macros
are  defined  by  POSIX  to allow the test of the file type in
st_mode to be written more concisely:

    S_ISREG(m)  is it a regular file?

    S_ISDIR(m)  directory?

    S_ISCHR(m)  character device?

    S_ISBLK(m)  block device?

    S_ISFIFO(m) FIFO (named pipe)?

    S_ISLNK(m)  symbolic link?  (Not in POSIX.1-1996.)

    S_ISSOCK(m) socket?  (Not in POSIX.1-1996.)
```

- `ENOENT` file/dir not found
- `EIO` - if `block_read` or `block_write` fail
- `ENOMEM` - out of memory
- `ENOTDIR` - not a directory
- `EISDIR` - is a directory
- `EINVAL` - invalid parameter (see comments in fs5600.c)
- `EOPNOTSUPP` - not implemented yet

Ex 2:2

helper: inode2stat(stat sb, fs_inode in, inode_num):

copy the info in an inode to struct stat

```
struct stat {
      ino_t     st_ino;         // Inode number
      mode_t    st_mode;        // File type and mode
      nlink_t   st_nlink;       // Number of hard links (set to 1)
      uid_t     st_uid;         // User ID of owner
      gid_t     st_gid;         // Group ID of owner
      off_t     st_size;        // Total size, in bytes
      blkcnt_t  st_blocks;      // Number of blocks allocated
                                // (note: block size is FS_BLOCK_SIZE;
                                // and this number is an int which should be round up)

      struct timespec st_atim;  // Time of last access (same as st_mtime)
      struct timespec st_mtim;  // Time of last modification
      struct timespec st_ctim;  // Time of last status change
  };
```

```
struct fs_inode {
    uint16_t uid;
    uint16_t gid;
    uint32_t mode;
    uint32_t ctime;      // time of l
    uint32_t mtime;      // last modi
    int32_t  size;
    uint32_t ptrs[NUM_PTRS_INODE]; /
};
```

Ex 2.3: fs_getattr()

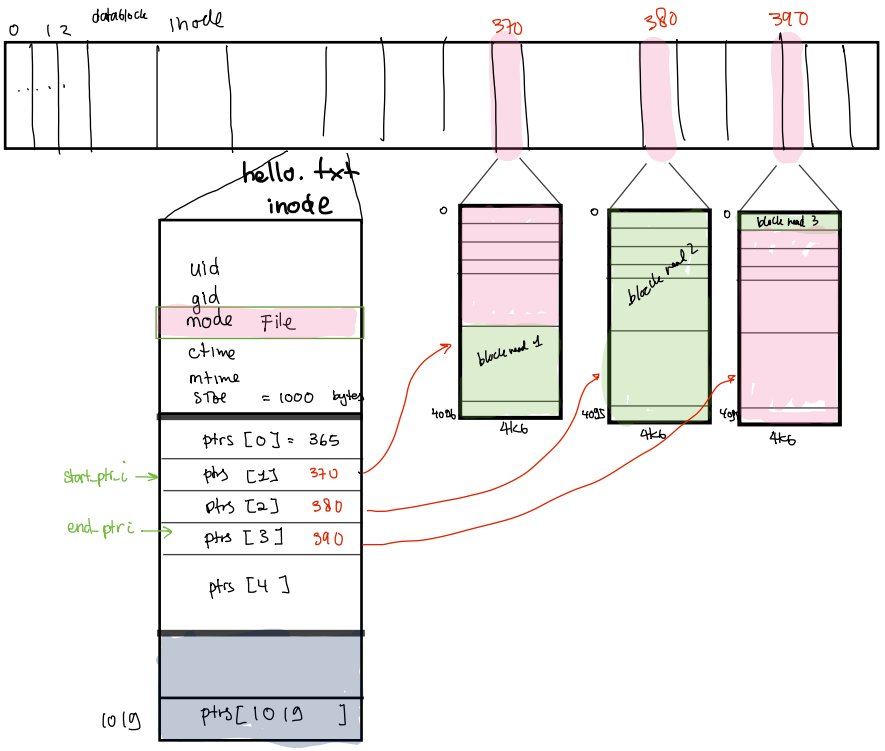
```c
int fs_getattr(const char *path, struct stat *sb)
{
    printf("\n>>>>getattr");


    // 1. get the inode num and inode
    int inodenum = path2inum(path);

    // check if inodenum returns error
    if (inodenum < 0) {
        return inodenum; // return the error code
    }

    // get the inode
    struct fs_inode curr_inode;
    if (block_read(&curr_inode, inodenum, 1) < 0) {
        free(path);
        return -EIO;
    }
    // 2. copy inodes info to struct stat
    inode2stat(sb, &curr_inode, inodenum);
    print_node_info(inodenum, curr_inode);
    print_stat(sb);

    // 3. return 0
    return 0;
}
```

Ex 2.4: fs_readdir()
get all the entries in directories. we will use ptr (the second argument) to fill this all the info

```c
int fs_readdir(const char *path, void *ptr, fuse_fill_dir_t filler,
               off_t offset, struct fuse_file_info *fi)
{
    // printf("\n>>>>readdir\n");
    // 1. get the inode num and inode
    // int dir_inodenum = path2inum(path);
    char *_path = strdup(path);
    int dir_inodenum = path2inum(_path);

    // 2. check if inodenum returns error
    if (dir_inodenum < 0) {
        return dir_inodenum; // return the error code
    }

    // 3. get the inode
    struct fs_inode dir_inode;
    if (block_read(&dir_inode, dir_inodenum, 1) < 0) {
        free(_path);
        return -EIO;
    }
    // 4. make sure this is a dir's inode
    if (!S_ISDIR(dir_inode.mode)) {
        free(_path);
        return -ENOTDIR;
    }

    // 5. copy the dir_entries to mem
    // get all its entries from disk (4096/32B = 128 entries) to memory
    int DIR_ENTRY_NUM = FS_BLOCK_SIZE / sizeof(struct fs_dirent);
    struct fs_dirent dir_entries[DIR_ENTRY_NUM];

    if (block_read(dir_entries, dir_inode.ptrs[0], 1) < 0) {
        free(_path);
        return -EIO;
    }
```

```c
    // 6. iterate trhough each entry
    for (int dir_entry_i = 0; dir_entry_i < DIR_ENTRY_NUM; dir_entry_i++) {
        if (dir_entries[dir_entry_i].valid == 1) {
            // 1. get the name of this entry
            char* entry_name = dir_entries[dir_entry_i].name;
            uint32_t entry_inodenum = dir_entries[dir_entry_i].inode;

            // 2. get the statbuf of this entry
            // - get inode
            struct fs_inode entry_inode;
            if (block_read(&entry_inode, entry_inodenum, 1) < 0) {
                free(_path);
                return -EIO;
            }

            // - use the inode to get statbuf
            struct stat entry_statbuf;
            inode2stat(&entry_statbuf, &entry_inode , entry_inodenum);

            // 3. fill
            filler(ptr, entry_name , &entry_statbuf, 0);

        }
    }
    // printf("finish iterating entries>>>>>>>>>>>\n");
    // return success
    return 0;
}
```

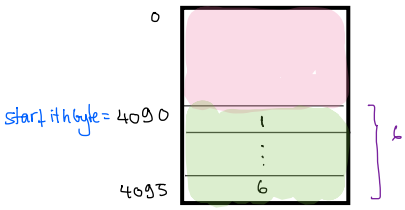
EX3.1 : fs_read()



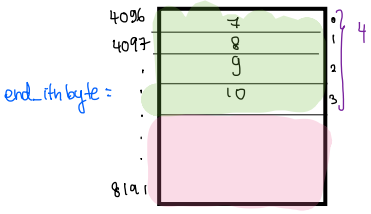
① validate

block read file inode

ptrs[0] =



ptrs[1] =



③ adjust length

total # ptrs = ceil (1000 / 4096)

we want to start from start_ith_bytes = 986th bytes of the data block

we want to read bytes_num_to_read = 17 bytes

so the end_ith_bytes = 986

= 1003 toobig!

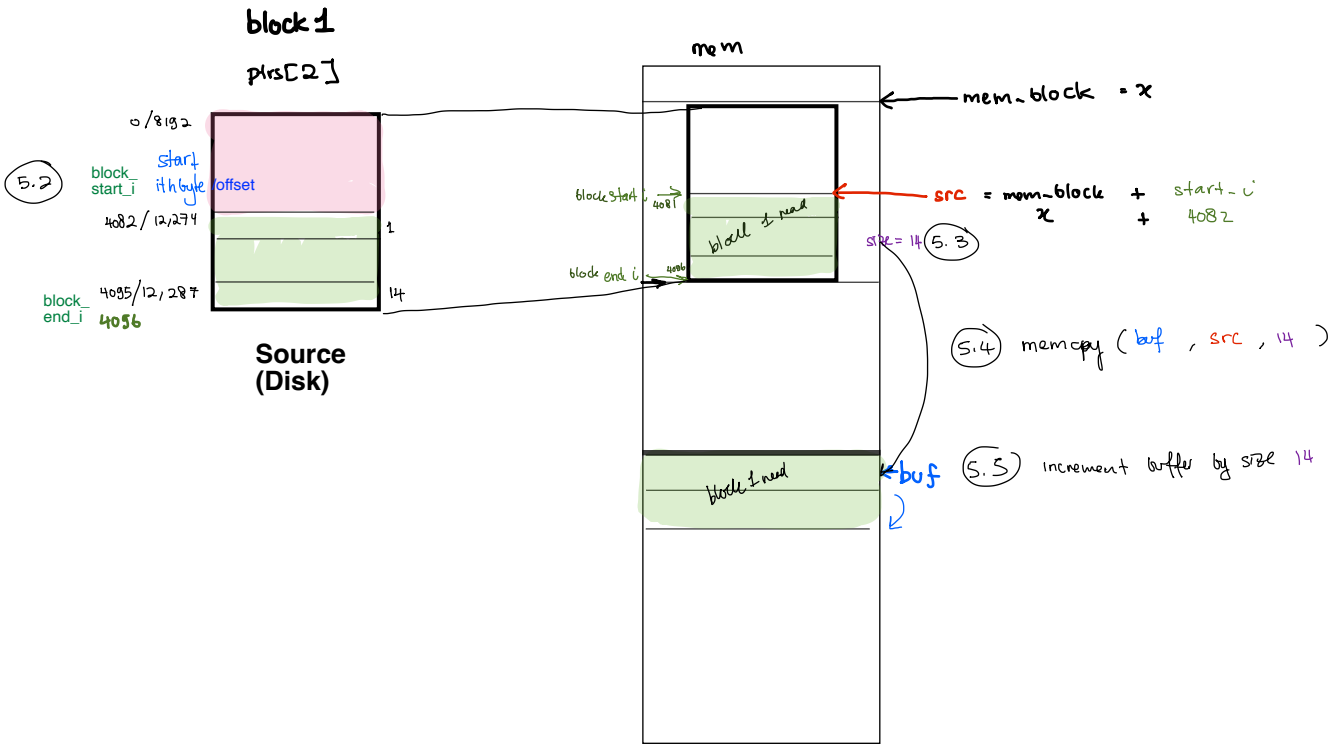
adjust length: bytes_num_to_read = 1000 - 986 = 14 bytes

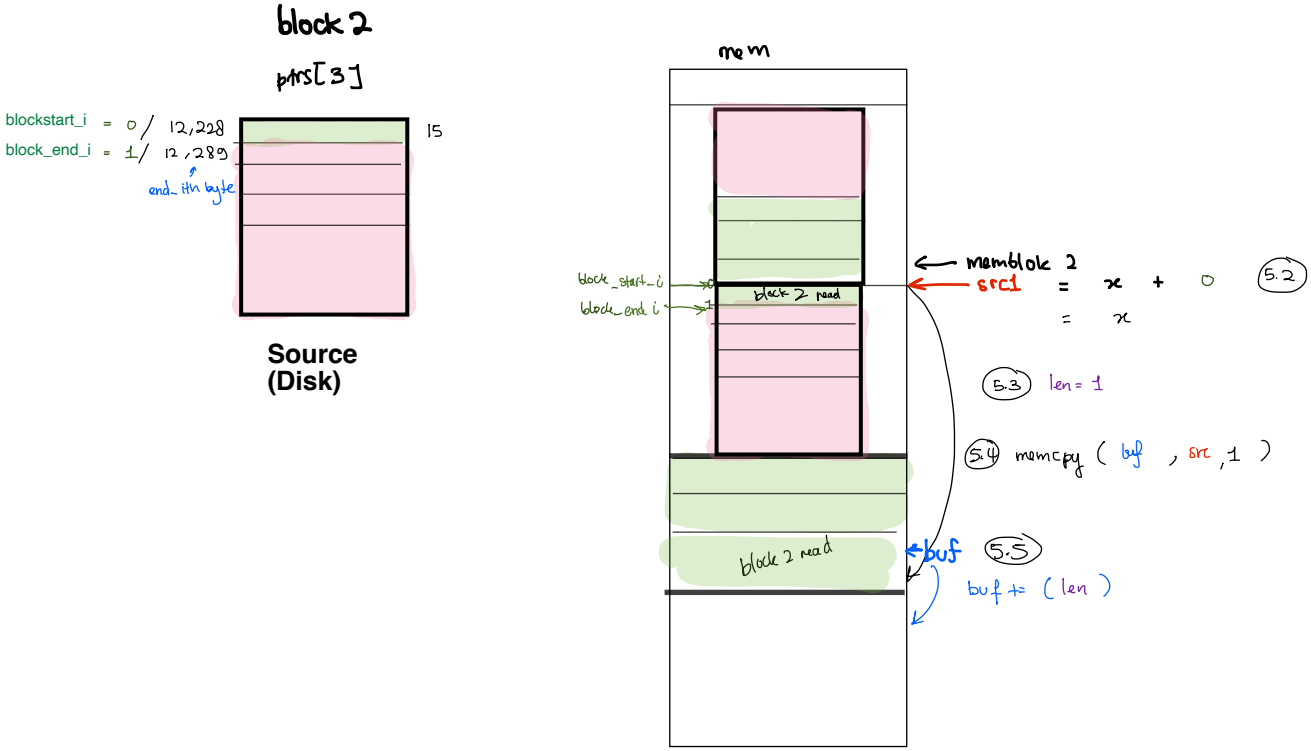
④ get the block number to read

start_ptr_i = start_ith_bytes / 4096

end_ptr_i = (start_ith_bytes + bytes_num_to_read - 1) / 4096

(5) read block





(6) read block fn

start_ptr_i — end_ptr_i : :

1 read block i

2 get block_start_i and block end i based on ith byte

3 get the length of this block

4. copy

5 increment buf ptr with length

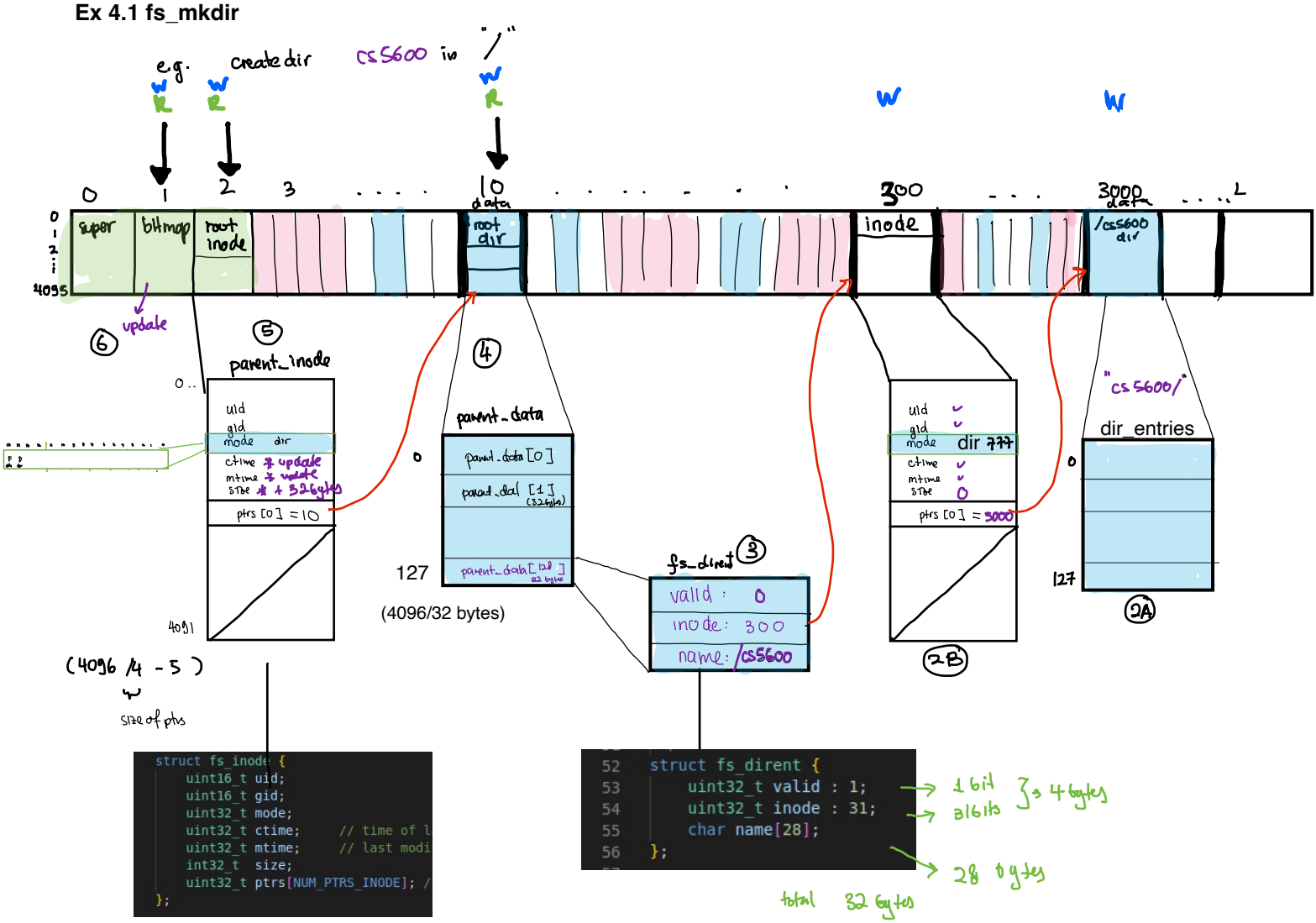
Note:
buf: This is the buffer into which the function will copy the data that it reads from the file. This buffer is provided by the caller of the function.

mem_block: This is a temporary buffer used within the function to store data read from the disk. The size of this buffer is FS_BLOCK_SIZE, which is a constant that defines the size of a block in the filesystem. When the function reads a block from the disk using block_read, it reads the entire block into mem_block. It then calculates which part of this block to copy into buf, based on the offset and the amount of data to be read. This could be the entire block or just a portion of it, depending on where the specified offset and length would have the read operation start and end.

. block_write( ) hints:

1. bitmap ↙ a new inode
2. inode # 10 ↙
3. datablock# 11 ↙ data (empty)
4. write to data block of root "/" dirents
5. modify the inode 2 :

Ex 4.1 fs_mkdir




```
struct fs_inode {
    uint16_t uid;
    uint16_t gid;
    uint32_t mode;
    uint32_t ctime;     // time of l
    uint32_t mtime;     // last modi
    int32_t  size;
    uint32_t ptrs[NUM_PTRS_INODE]; /
};
```

```
52  struct fs_dirent {
53      uint32_t valid : 1;
54      uint32_t inode : 31;
55      char name[28];
56  };
```

1 bit } 4 bytes
31 bits
28 bytes
total 32 bytes

part 0 : validate

part 1 : find 2 empty blocks from bitmap

1. read_block (bit_block, block num 2)
2. get 2 bits that are 0 :
   - 1 for new dirs inode num
   - 1 for new dirs datablock num

Part 2 inode & data for cs5600

- (A) • init empty data
- (B) • inode :→ uid → ctime
  → gid → mtime
  → mode → size :0
  → ptrs[0] = new dirs datablock num

Part 3. fill in the fs_dirent of cs5600

valid = 1
inode = new dirs inode num
name = cs5600

Part 4 data for parent update

1. read_block the parent node (to mem)
2. read_block the parent data (to mem) par_dir_entries
3. find "invalid" entry in mem
   → assign cs5600 fs_dirent to this index

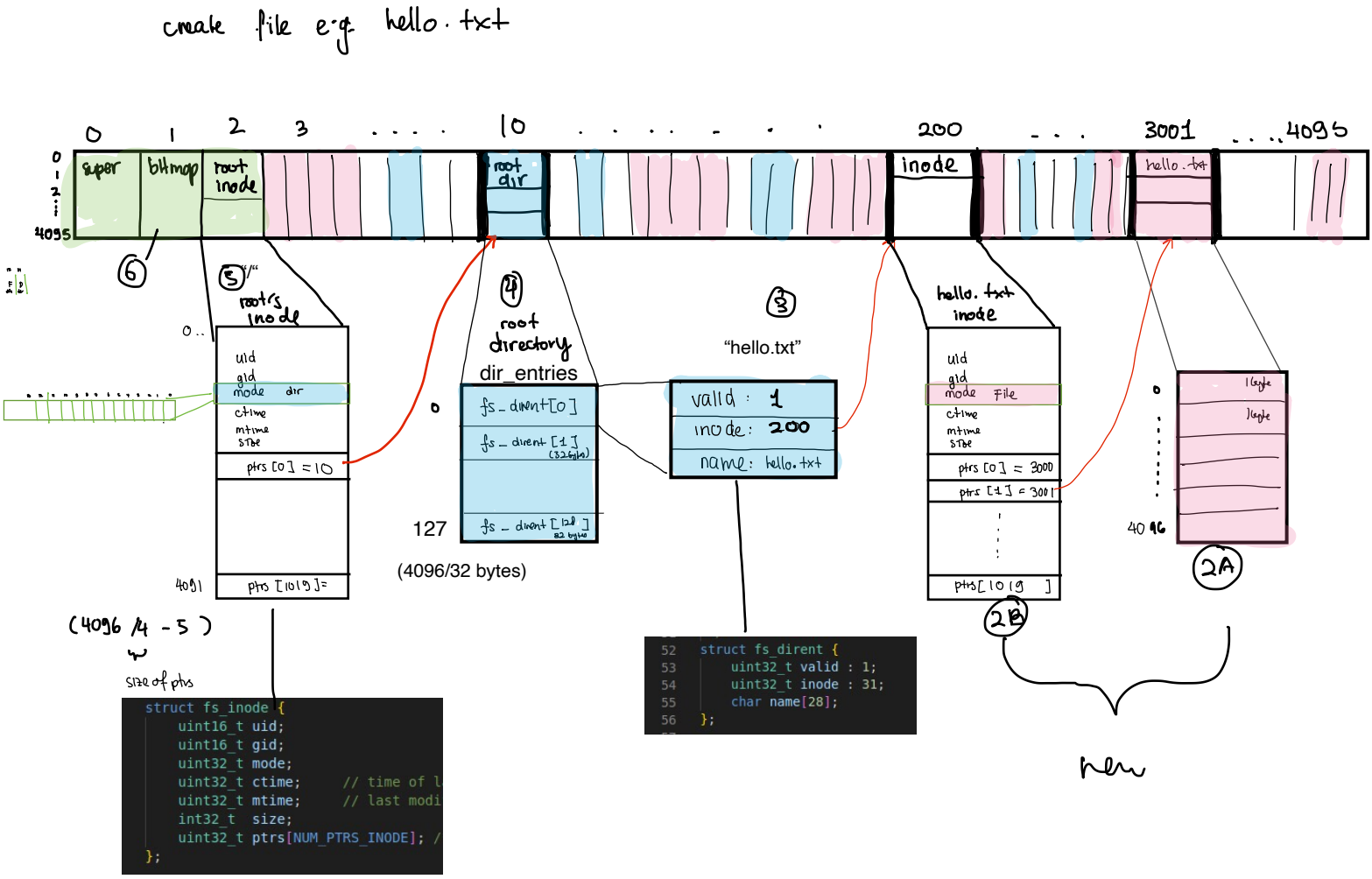
Part 5 node of parent update

Part 6 update bitmap set to 1
on newdir_inode_num
newdir_datablock_num

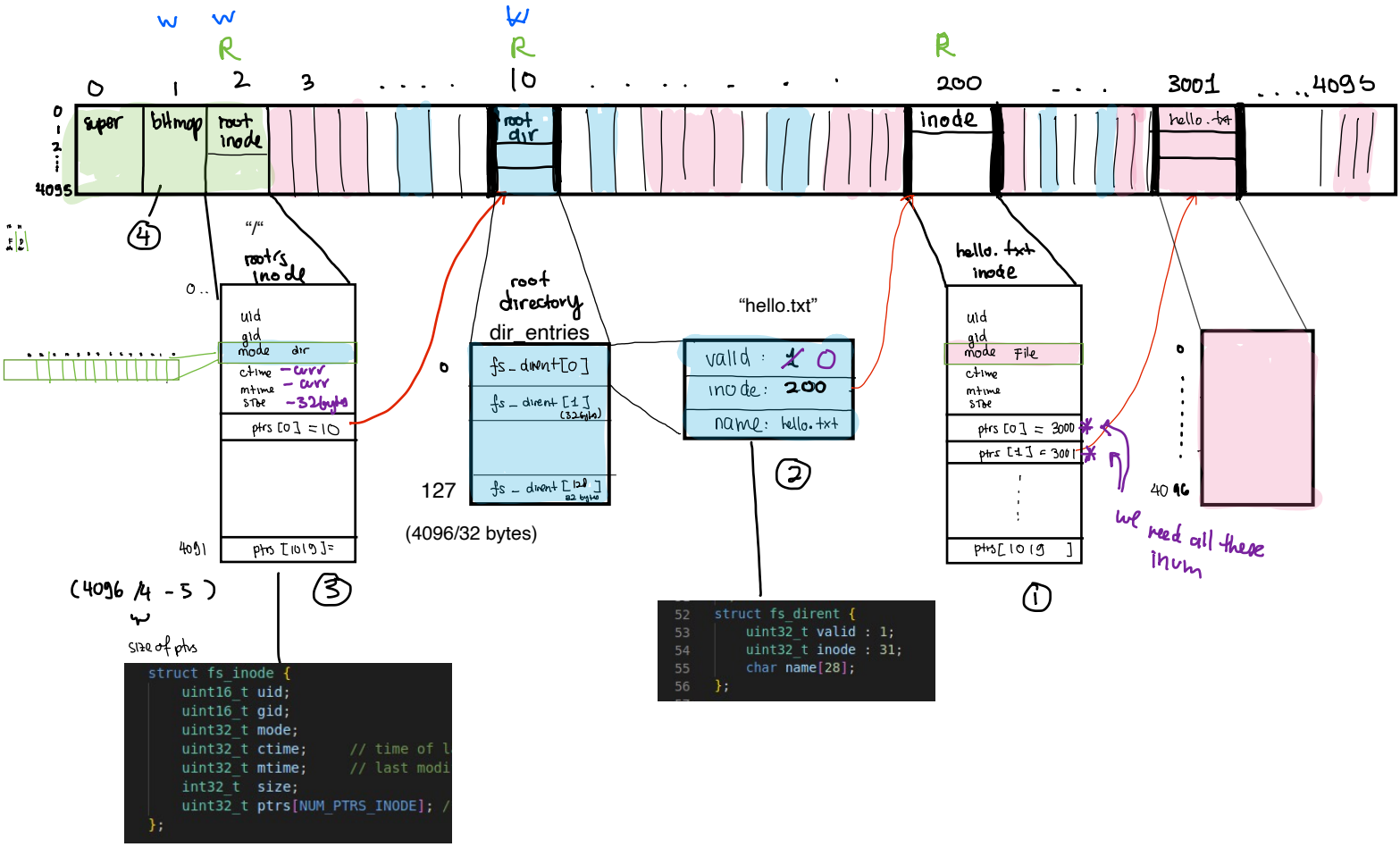
Part 7

write block: parent inode
parent data (direntry)
cs5600 inode
cs5600 data (direntry)
bitmap

Ex 4.2 fs create



Ex 5.1 fs_unlink (delete a file)




```
struct fs_inode {
    uint16_t uid;
    uint16_t gid;
    uint32_t mode;
    uint32_t ctime;      // time of l
    uint32_t mtime;      // last modi
    int32_t  size;
    uint32_t ptrs[NUM_PTRS_INODE]; /
};
```

```
52  struct fs_dirent {
53      uint32_t valid : 1;
54      uint32_t inode : 31;
55      char name[28];
56  };
```

part 0 validate and

make sure we have block_read these:

- parent_inode
- parent_data
- file_inode

part 1 get info from file's mode

- store all the inum in int[] inums_delete;
- free blk

part 2 update parent's datablock

in data block of parent, find the filename,

- set valid to 0
- add the inum to inums_del;
- free blk

part 3 update parent's inode

update ctime and mtime, decr size by -32bytes

part 4: update block bitmap

part 5 blockwrite

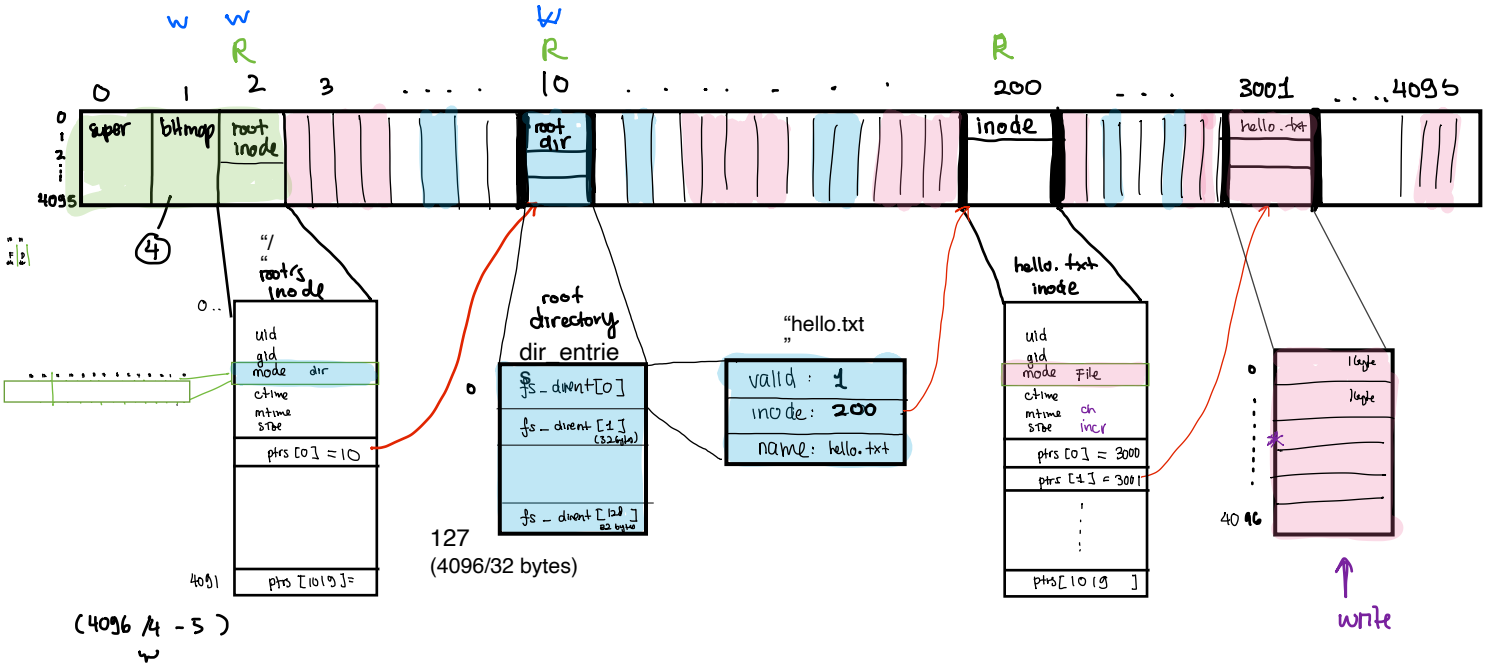
blockwrite : bitmap
parent inode
parent_data

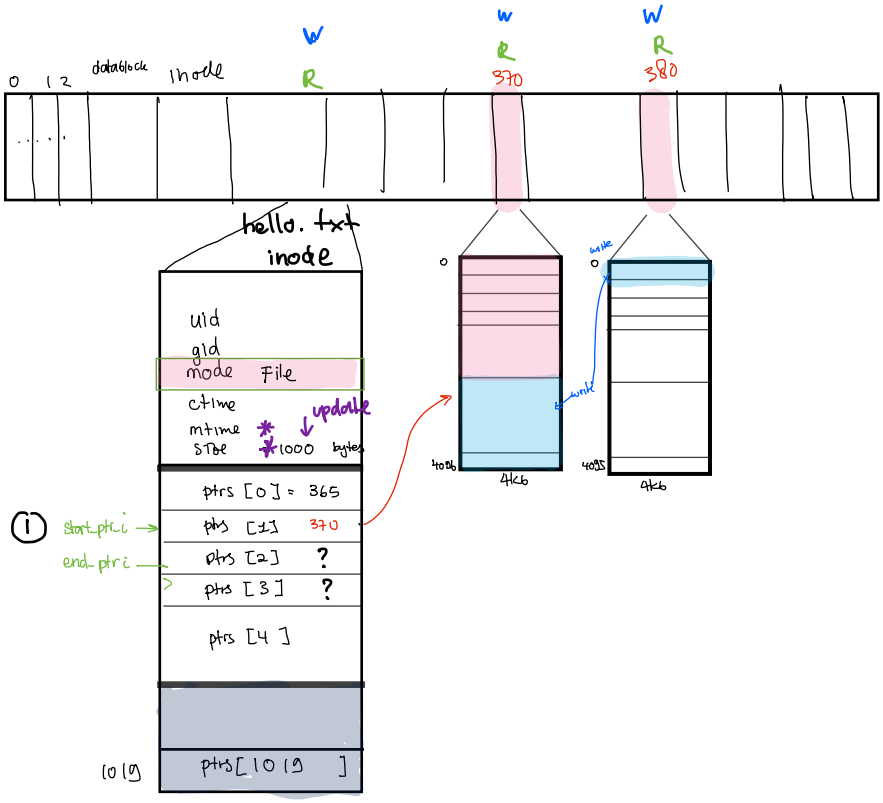
EXERCISE 5 implement `fs_unlink` and `fs_rmdir`.

- implement `fs_unlink`, removing a file
- `fs_unlink` errors:
  - bad path `/a/b/c` - `b` doesn't exist (`-ENOENT`)
  - bad path `/a/b/c` - `b` isn't directory (`-ENOTDIR`)
  - bad path `/a/b/c` - `c` doesn't exist (`-ENOENT`)
  - bad path `/a/b/c` - `c` is directory (`-EISDIR`)
- implement `fs_rmdir`, removing a directory
- `fs_rmdir` errors:
  - bad path `/a/b/c` - `b` doesn't exist (`-ENOENT`)
  - bad path `/a/b/c` - `b` isn't directory (`-ENOTDIR`)
  - bad path `/a/b/c` - `c` doesn't exist (`-ENOENT`)
  - bad path `/a/b/c` - `c` is file (`-ENOTDIR`)
  - directory not empty (`-ENOTEMPTY`)
- after implementing, you should pass another 10 cases:

```
$ make testb
...
51%: Checks: 27, Failures: 13, Errors: 0
```

Ex 6.1 fs write to a file.





Part ⓪ validate

block read — file inode

file data

① file_num not found (enoent) — args: 1. inum

② file_inode is a dir (eisdir) — 2. inode

③ offset = start_ith_bytes > curr file length (EINVAL) — 3. start_ith_bytes; max file size

④ bytes_num_to_write + file_inode.size > (1019*4kb) — 4. len

ENOSPC

Part ① get start_ptr_i and end_ptr_i we can read all the request block number

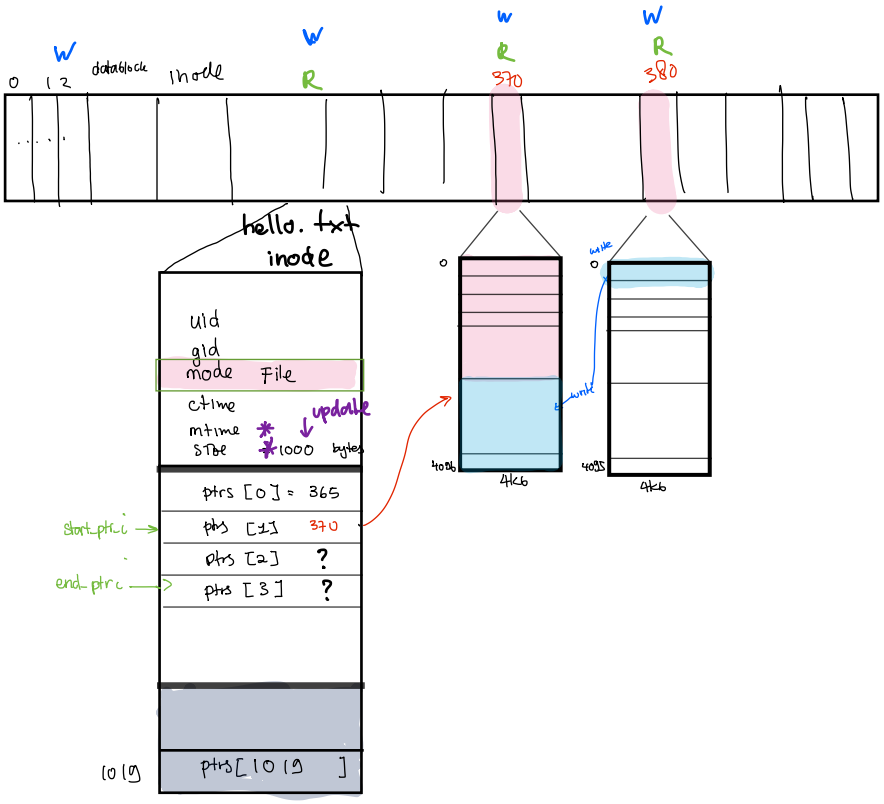
---

end_ith_bytes = start_ith_bytes + bytes_num_to_read (excl.)

start_ptr_i = start_ith_bytes (7000) / 4096 = 1

end_ptr_i = (end_ith_bytes (excl.) (9000) − 1) / 4096 = 2

Part 2. Check Block by Block if data block that we want to write to already exist.



if Data inum Invalid check if His bitset to 0 (which means we can block read this)

int data_inum = file_inode.ptrs[i]

// Case A: already exist, just load to mem

if (data_inum ≥ 3 and data_inum < num_blocks)

and bitset (block_bitmap, data_inum) {

mem dst, disk src

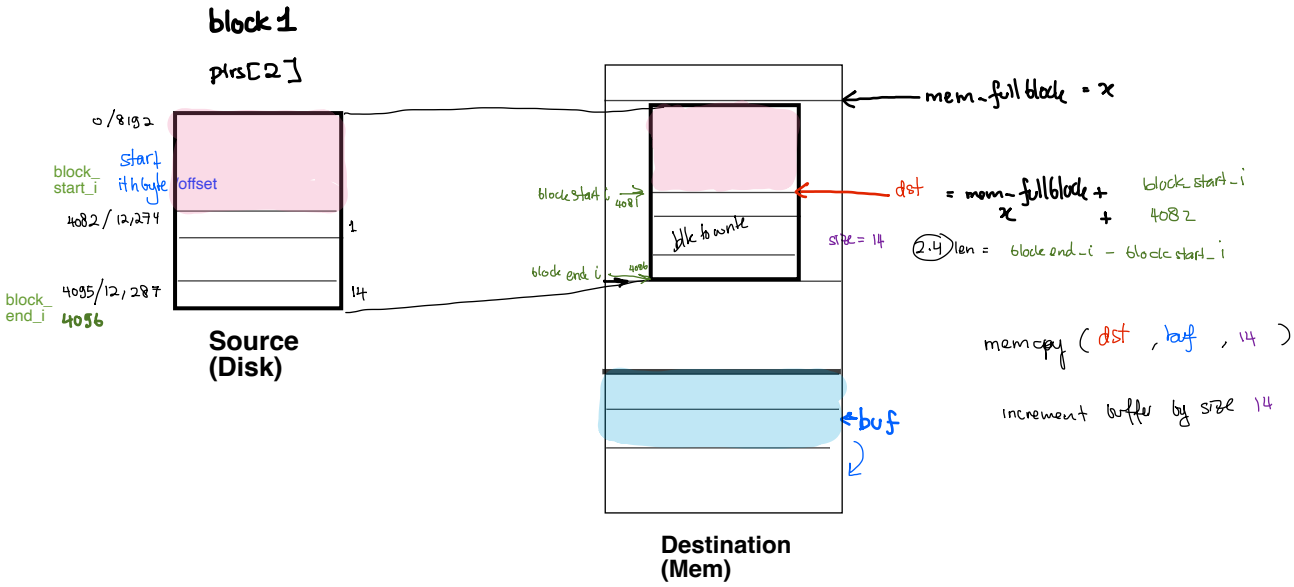
block read (mem_full_block, data_inum, 1)

}

// case B. not yet exist, init new datablock

- int new_datablock num = alloc_blk()
- insert this to file_inode ptrs
- init memblock in memory.

Part 3. Write block by block.



mem dst

disk src

1 block read (mem_full_block , file_inode . ptrs [i] , 1)

2 get the block_start_i to know when to start writing.

if this is the first block :

ptr_i == start_ptr_i

block_start_i = start_ith byte % FS_BLOCK SIZE

else : 0

3. get the block_end_i (exclusive) to know when to finish writing:

if this is the last block and the last byte is to write is not the end of the block

ptr_i == end_ptr_i && end_ith byte % FS_BLOCK SIZE != 0

block_end_i = end_ith byte % FS_BLOCK SIZE

else :

block_end_i = FS_BLOCK_SIZE

4. get the length of data to write

len_write_pblock = block_start_i - block_end_i

5. get dst address in memblock full where to write

dst = mem_full_block + block_start_i

6. block write this data from buffer mem_full_block to no file_inode.ptrs[i]

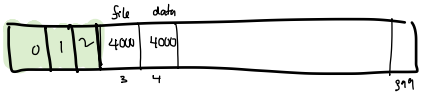
7. incr. buf

fs_truncate:

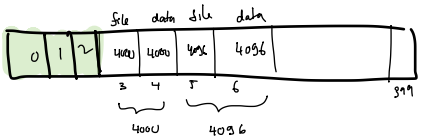
file_4000

write :

truncate :



file_4096



Chall. caesar

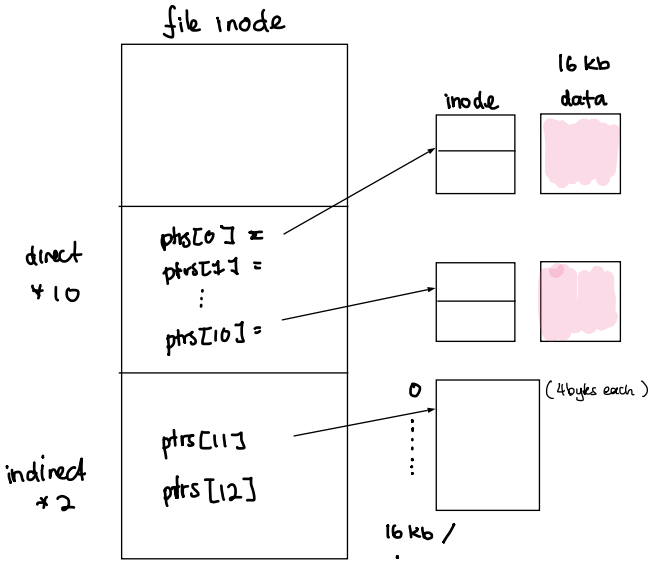
1. get_caesar_key() will return the key if file exist.

2. fs_init():

```
// to test:
// fs_create("/key");
// fs_write("2");
// fs_create("abc.txt");
// fs_write("abc");
```

2. fs_read(path, buf, len, offset, fi):

```
//1.init caesar key
int caesar_key = 0;
if path is not "/key":
    caesar_key = get_caesar_key()

// 2. when we read
```



Should be able to point to all the 2^{20} of 16 kb means it should be able to store 2^{20} addresses.