

1. a) 1) 4 - inode blocks. 1 for the file c, and 3 for the directories /, a, b
  - 2) 3 - directory blocks - one for root /, one for a, the other for b
  - 3) 1 - single indirect block as far as we know. The file definitely has more than 12 blocks (# of data blocks pointed by direct pointers), but less than 1036 (# of data blocks pointed by direct pointers and single indirect pointers). We are reading block 1034.
  - 4) 1 - data block for file c
2. (a) All of the above

### Notes

#### • Inode



- Is short form of **index node**
- Describes a file system object such as file or data
- Contains all information about a file/directory, including
  - \* File Type,
  - \* Size
  - \* Number of blocks allocated to it
  - \* Protection information
  - \* Time information (e.g time created, time modified)
  - \* Location of data blocks residing on disk

### References

- 1) Wikipedia, Inode, link
  - 2) Machanick, Philip. (2016). Teaching Operating Systems: Just Enough Abstraction. 642. 10.1007/978-3-319-47680-3\_10., link
- (b) Size, the location of data blocks that reside on disk

### Notes

- I wonder what information about blocks inode has. Is it total number of blocks both inode and data, or just data?
- I struggled a bit on this one. I should find an easier way to remember which information inode has
- (c) • **Inode Bitmap and Data Block Bitmap**
  - (b) - Data Leak
  - (c) - Inode Leak
- **New Directory Inode**
  - (a) - No inconsistency
- **Inode Bitmap, Data Block Bitmap, Existing Directory Data, New Directory Inode, and New Directory Data**
  - (e) - Inconsistent inode data
- **Inode Bitmap, and New Directory Inode**
  - (c) - Inode leak
  - (d) - Multiple file paths may point to same inode
- **New Directory Inode, Existing Directory Inode, Existing Directory Data**
  - (e) - Inconsistent inode data
  - (f) - Something points to garbage

### Correct Solution

- **Inode Bitmap and Data Block Bitmap**
  - (b) - Data Leak
  - (c) - Inode Leak
- **New Directory Inode**
  - (a) - No inconsistency
- **Inode Bitmap, Data Block Bitmap, Existing Directory Data, New Directory Inode, and New Directory Data**
  - (e) - Inconsistent inode data
- **Inode Bitmap, and New Directory Inode**
  - (c) - Inode leak
- **New Directory Inode, Existing Directory Inode, Existing Directory Data**

(d) - Multiple file paths may point to same inode  
 (f) - Something points to garbage

### Notes

- I wonder how system call for reading file/directory works in UNIX. Does it check for bitmap?
- I wonder how system call for deleting file/directory works in UNIX
- I wonder how system call for creatubg file/directory works in UNIX
- Learned that
  - Missing Inode Bitmap - multiple file paths may point to same inode

### • **File API**

- open (create/access file)
  - \* Is a system call
  - \* Reads target inode into memory (when loading)
  - \* Does three things on creation
    - 1) make structure (inode) that racks all relevant information about file
    - 2) link human readable name to the file, and put that link to a directory
    - 3) increment **reference count** in inode

#### \* **Syntax:**

```
int fd = open("foo". O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR)
```

- O\_CREAT - Creates file "foo" if does not exist
- O\_WRONLY - Open file for writing only (default)
- O\_TRUNC - Overwrites existing file **Need example/Clarification**
- Can have multiple flags
- \* Returns **file descriptor** or `fd` for short
  - Is an integer
  - Is used to access a file
  - Is private per process
  - Can be used to `read()` and `write()` files

### Example

```

#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *filename = "/tmp/file";
...
fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, mode);
...

```

File can be read by owner (points to S\_IRUSR)

File can also be written by owner (points to S\_IWUSR)

File can also be read by group (points to S\_IRGRP)

File can also be read by others (points to S\_IROTH)

Means

1. File is Writable AND
2. Create file if doesn't exist AND
3. Overwrite file if exists

- \* Amount of I/O generated by `open()` is proportional to length of pathname (wait. How is I/O involved in `open()`?)

– (read) (read file)

- \* Is a system call

\* **Syntax:**

```
ssize_t read (int fd, void *buf, size_t count)
```

- `fd` - file descriptor (from `open()`)
- `buf` - container for the read data
- `count` - number of bytes to read

- \* Returns number of bytes read, if successful
- \* Returns 0 if is at, or past the end of file

### Example

```

char buf[4096];
int fd = open("/a/b/c", O_RDONLY); // open in read-only mode
lseek(fd, 1034*4096, 0); // seek to position (1034*4096) from start of file
read(fd, buf, 4096); // read 4k of data from file

```

System Calls	Return Code	Current Offset	
<code>fd = open("file", O_RDONLY);</code>	3	0	
<code>read(fd, buffer, 100);</code>	100	100	read continues for each call
<code>read(fd, buffer, 100);</code>	100	200	
<code>read(fd, buffer, 100);</code>	100	300	
<code>read(fd, buffer, 100);</code>	0	300	
<code>close(fd);</code>	0	-	returns 0 if at end

– write (write file)

- \* Is a system call
- \* Writes data out of a buffer
- \* **Syntax:**

```
ssize_t write (int fd, const void * buf, size_t nbytes)
```

- fd - file descriptor
- buf - A pointer to a buffer to write to file
- nbytes - number of bytes to write. If smaller than buffer, the output is truncated

### Example

```
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
    int filedesc = open("testfile.txt", O_WRONLY | O_APPEND);

    if (filedesc < 0) {
        return -1;
    }

    if (write(filedesc, "This will be output to testfile.txt\n", 36) != 36) {
        write(2, "There was an error writing to testfile.txt\n", 43);
        return -1;
    }

    return 0;
}
```

– lseek

- \* Reads or write to a specific offset within a file
- \* **Syntax:**

```
off_t lseek (int fd, off_t offset, int whence)
```

- fd - file descriptor
- offset - the offset of pointer within file (in bytes)
- whence - the method of offset

SEEK\_SET - offset from the start of file (absolute)

SEEK\_CUR - offset from current location + offset bytes (relative)

SEEK\_END - offset from the end of file

- \* Returns offset amount (in bytes) from the beginning of file
- \* Returns -1 if error

### Example

System Calls	Return Code	Current Offset
<code>fd = open("file", O_RDONLY);</code>	3	0
<code>lseek(fd, 200, SEEK_SET);</code>	200	200
<code>read(fd, buffer, 50);</code>	50	250
<code>close(fd);</code>	0	-

- rename (update file name)
  - \* Is a system call
  - \* Changes the name of file
  - \* Is **atomic** (after crash, it will be either old or new, but not in-between)
  - \* **Syntax:** `int rename(const char *old, const char *new)`
    - old - name of old file
    - new - name of new file
  - \* Returns 0 if successful
  - \* Returns -1 if error

### Example

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

- stat (get file info)
  - \* displays metadata of a certain file stored in **inode**
  - \* **Syntax:** `int stat(const char *path, struct stat *buf)`
    - path - file descriptor of file that's being inquired
    - buf - A stat structure where data about the file will be stored (see below)

```
struct stat {
    dev_t    st_dev;      // ID of device containing file
    ino_t    st_ino;      // inode number
    mode_t    st_mode;    // protection
    nlink_t   st_nlink;   // number of hard links
    uid_t     st_uid;     // user ID of owner
    gid_t     st_gid;     // group ID of owner
    dev_t     st_rdev;    // device ID (if special file)
    off_t     st_size;    // total size, in bytes
    blksize_t st_blksize; // blocksize for filesystem I/O
    blkcnt_t  st_blocks;  // number of blocks allocated
    time_t    st_atime;   // time of last access
    time_t    st_mtime;   // time of last modification
    time_t    st_ctime;   // time of last status change
};
```

Figure 39.5: The **stat** structure.

### Example

```
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char **argv)
{
    if(argc != 2)
        return 1;

    struct stat fileStat;
    if(stat(argv[1], &fileStat) < 0)
        return 1;

    printf("Information for %s\n", argv[1]);
    printf("-----\n");
    printf("File Size: \t\t%d bytes\n", fileStat.st_size);
    printf("Number of Links: \t\t%d\n", fileStat.st_nlink);
    printf("File inode: \t\t%d\n", fileStat.st_ino);

    printf("File Permissions: \t");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n\n");

    printf("The file %s a symbolic link\n", (S_ISLNK(fileStat.st_mode)) ? "is" : "is not");

    return 0;
}
```

The result of above is:

```
$ ./testProgram testfile.sh

Information for testfile.sh
-----
File Size:                36 bytes
Number of Links:          1
File inode:                180055
File Permissions:         -rwxr-xr-x

The file is not a symbolic link
```

- unlink (removing file)
  - Is a system call
  - Removes a file (including symbolic link) from the system
  - **Syntax:** `int unlink(const char *pathname)`
    - \* pathname - path to file
  - Returns 0 if successful
  - Returns -1 if error

### Example

```
#include <unistd.h>

char *path = "/modules/pass1";
int status;
...
status = unlink(path);
```

- `mkdir` (creating directory)
  - Is a system call
  - **Syntax:** `int mkdir(const char *path, mode_t mode)`
    - \* `path` - path of directory (including name)
    - \* `mode` - permission group
  - Returns 0 if successful
  - Returns -1 if error
  - directories can never be written directly
    - \* directory is in format called **File System Metadata**
    - \* directory can only be updated directly
  - creates two directories on creation `.` (current) and `..` (parent)

### Example

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkdir("/home/cnd/mod1", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
```

- `opendir`, `readdir`, `closedir` (reading directory)
  - Are system calls
  - Are under `<dirent.h>` library
  - Requires struct `dirent` data structure

```
struct dirent {
    char          d_name[256]; // filename
    ino_t         d_ino;       // inode number
    off_t         d_off;       // offset to the next dirent
    unsigned short d_reclen;    // length of this record
    unsigned char  d_type;      // type of file
};
```

- **Syntax (`opendir`):** `DIR *opendir(const char *dirname)`



- \* dirname - directory path
- \* Returns a pointer to the directory stream
- \* The stream is positioned at the first entry in the directory.
- **Syntax (readdir):** `struct dirent *readdir(DIR *dirp);`
  - \* dirp - directory stream
  - \* Returns a pointer to a dirent structure representing the next directory entry in the directory stream
  - \* Returns NULL on reaching the end of the directory stream
- **Syntax (closedir):** `int closedir(DIR *dirp);`
  - \* dirp - directory stream
  - \* Returns 0 if successful
  - \* Returns -1 otherwise

### Example

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino,
              d->d_name);
    }
    closedir(dp);
    return 0;
}
```

- rmdir (Deleting Directories)
  - Removes a directory whose name is given by path
  - Is performed only when directory is empty
  - Is included in <unistd.h> library
  - Fails if is symbolic link
  - **Syntax:** `int rmdir(const char *path)`
    - \* path - path of directory
  - Returns 0 if successful
  - Returns -1 if error

### Example

```
#include <unistd.h>

int status;
...
status = rmdir("/home/cnd/mod1");
```

- `unlink` (Remove file)
  - Remove a link to a file
  - Is called **unlink** because it decrements **reference count** in inode
    - \* Deletes file completely when reference count within the inode number is 0
  - **Syntax:**

```
#include <unistd.h>

int unlink(const char *pathname);
```

\* `pathname` - `pathname` to file

- Returns 0 if successful
- Returns -1 if error
- Is used by linux command `rm`

### Example

```
#include <unistd.h>

char *path = "/modules/pass1";
int status;
...
status = unlink(path);
```

```

prompt> echo hello > file
prompt> stat file
... Inode: 67158084      Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084      Links: 2 ...
prompt> stat file2
... Inode: 67158084      Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084      Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084      Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084      Links: 1 ...
prompt> rm file3

```

- **Symbolic Link:**

- Is directory entry containing "true" path to the file
- Is a shortcut that reference to a file instead of inode value <sup>[2]</sup>



2



3



- **Hard Link:**

- Is a direct reference to a file via its inode <sup>[2]</sup>
- Is second directory entry identical to first

1



2



3



4



- Crash Consistency
  - Inode before update

```

owner      : remzi
permissions : read-write
size       : 1
pointer    : 4
pointer    : null
pointer    : null
pointer    : null

```



– Inode after update

```

owner      : remzi
permissions : read-write
size       : 2
pointer    : 4
pointer    : 5
pointer    : null
pointer    : null

```



### References

- 1) codewiki, stat, link
  - 2) The Open Group Base Specification, unlink, link
3. a) Indexed-based file system uses inode number and pointers to find data blocks, and data blocks can be set and anywhere, so external fragmentation isn't a problem
- b) Extent based file system only requires a pointer to first data block of file, and the rest is read by traveling contiguously, and this requires less disk block access than indexed-based file system of which has to go to inode block, indirect pointers and data blocks to go to a particular byte in file.

### Notes

- What is a sector? What is a sector address?
- How can I get to specific inode from block (e.g inode # 32 on block 2)?
- I should record differences between linked-list-based FS, Extent-based FS, and indexed FS
- **Index Based File System**



- Has 15 blocks of pointers that points to either inode, indirect pointers, or data block
- No external fragmentation
- Files can be easily grown

### Example

Linux's ext2, ext3

- **Extent Based File System**



- Requires only a disk pointer + length (in blocks)
- Is also called **contiguous allocation**
- Stores starting block and total length

- Is less flexible but more compact
- Works well when there is enough free space on the disk and files can be laid out contiguously

### Example

Linux's ext 4

- Advantage
  - \* Is simple
    - Finding data block = beginning of data block + length
- Disadvantage
  - \* Growing file size could cause problems
  - \* Fragmentation

#### • inode

- Inode block computation

$$\text{block number} = (\text{inode \#} * \text{sizeof}(\text{inode})) / \text{block size} \quad (1)$$

### Example

Target: inode #32

Inode Size: 256 bytes

Block Size: 4096 bytes

$$\text{block number} = (\text{inode \#} * \text{sizeof}(\text{inode})) / \text{block size} \quad (2)$$

$$= \frac{32 * 256}{4096} \quad (3)$$

$$= 2 \quad (4)$$

#### • superblock

- Contains information about the following
  - \* The number of inodes and data blocks in a particular file system
  - \* The magic number of some kind to identify the file system type
  - \* Where the inode table begins
- Is read first on mount before attaching to file system

#### • inode/data bitmap

- Accessed only when allocation/deallocation is needed
  - \* Read() → no bitmap required
- Uses bit to indicate whether the corres object/block is free



- \* 0 means free
- \* 1 means in use

## • Reading a File from Disk

### Example

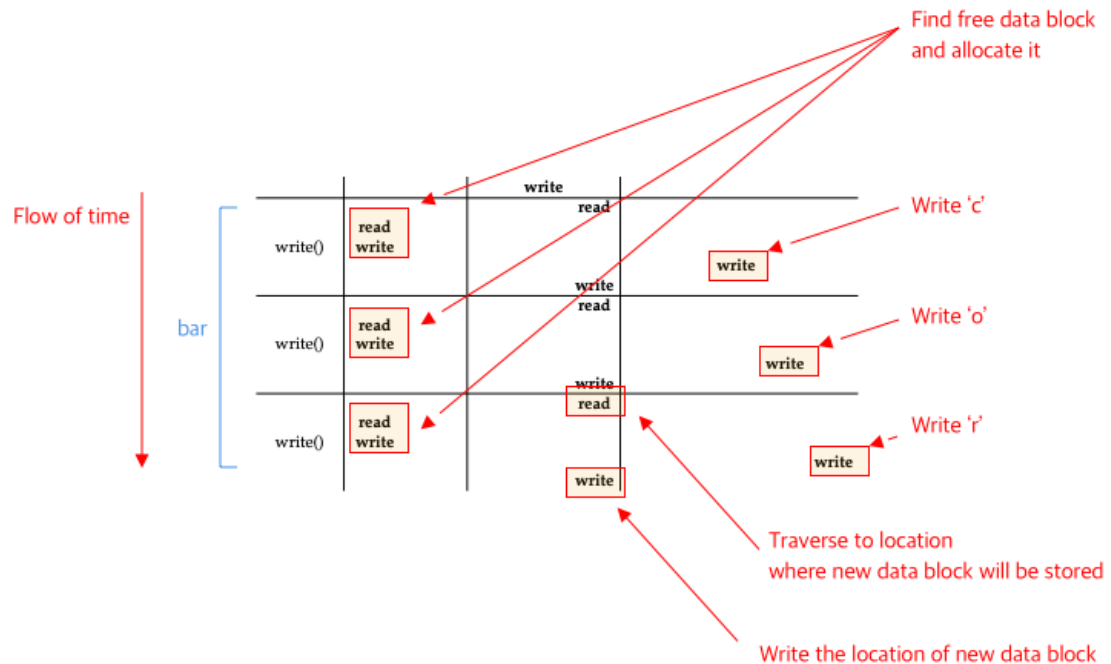
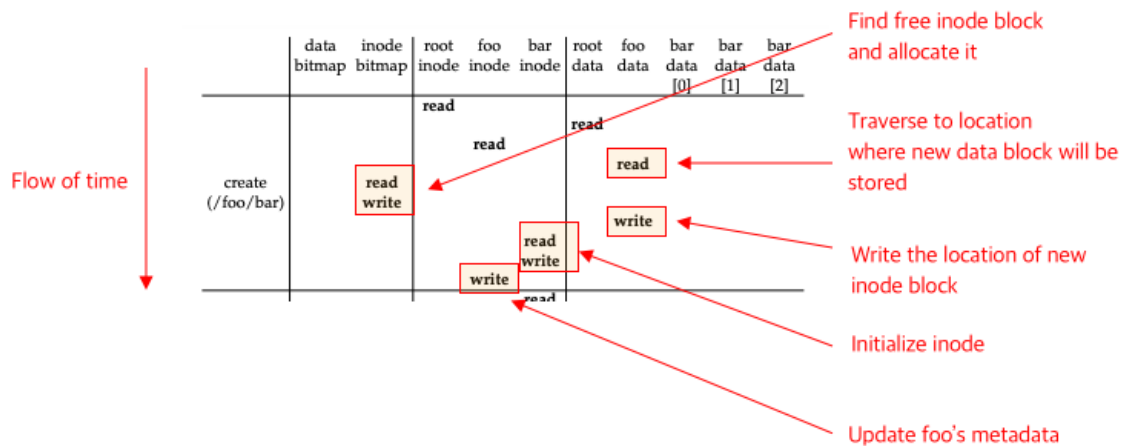
When

```
open("/foo/bar", O_RDONLY)
```

is called

- the goal is to find the inode of the file `bar` to read its basic information (i.e. includes permission, information, file size etc)
- done by traversing the pathname and locate the desired inode
- Steps
  1. Find **inode** of the root directory by looking for **i-number** (or **inode number**)
    - \* Root directory has no parent directory
    - \* Root directory's **inode number** is 2 (for UNIX file systems)
  2. Read the **inode** of root directory
  3. Once its **inode** is read, read through its directory data (pointers to **data blocks**) until the inode number of `foo` is found (e.g 42)
  4. Recursively traverse the pathname until the desired inode is found (more specifically, the **inode number** of `bar`)
  5. Issue a `open()` to read `bar`'s inode to memory
  6. Issue a `read()` system call to read from file `bar`
    - \* without `lseek()`, reads file from the first file data block (e.g. `bar data[0]`)
    - \* `lseek(..., offset_amt * size_of_file_block)` is used to offset/move to desired block in `bar`
  7. Transfer data to `buf` data block
  8. Read until `read()` returns 0, or desired data block has been read
  9. Close `fd`. No I/O is read.

## • Writing to Disk



Given a call

`create(...)` (Note: open to be exact)

- 5 I/Os are generated per write
  - \* Read inode (to traverse to the location of new data block)
  - \* Reading data bitmap
  - \* Writing data bitmap
  - \* Write data block

- \* Write inode (to update data block's location in inode)
- 10 I/Os are generated per file creation:
  - \* Read inode bitmap (to find free inode)
  - \* Write inode bitmap (to mark it allocated)
  - \* Create one new inode (to initialize it)
  - \* Write the location of new inode block in `foo` (by linking high-level name of file `bar` to its inode number and storing in data block)
  - \* Perform one read and write to the directory inode and update it

- **Static Partitioning**

- Divides resources into fixed proportion once
  - \* e.g. two possible users of memory → give fraction of memory to one user and rest to the other
- Advantages
  - \* Ensures each user receives some share of the resource
  - \* Delivers more predictable performance (usually)
  - \* Easier to implement
- Disadvantages
  - \* Is wasteful
  - \*

- **Dynamic Partitioning**

- Gives out different amounts of resources over time
- Lets resource-hungry users consume idle resources
- Advantages
  - \* Flexible
  - \* Can achieve better utilization than **static partitioning**
- Disadvantages
  - \* More complex to implement
  - \* Could lead to worse performance
    - e.g idle resource got consumed by others and take long time to reclaim it when needed (the periodic frozen feeling when loading screen)

## References

1) Columbia University, Operating Systems, link

4. 1) i) all of the above  
 ii) To minimize damage, it should be updated in the order of

1. Data Region
2. Inode Table
3. Data Bitmap
4. Inode Bitmap

The reason is that when data block and/or inode block are set and crash happens, the file system treats as if nothing had happened.

And the reason is that before data/inode block is allocated, it first checks the inode and data bitmap, which contains information about whether the block is occupied or not.

Once they are allocated, we risk data/inode leak, and if done improperly, the data/inode block would not be available until actions are taken.

- 2) i) inode bitmap, data bitmap
- ii) To minimize damage, it should be updated in the order of

1. Data Bitmap
2. Inode Bitmap

The reason is that when data bitmap is removed and crash happens, The data are still in place, and those can be used to continue the operation (assuming that nothing is disturbed).

More specifically, deletion of file is complete when reference count in the inode of file B hits 0, and as long as data is in place we can work so that the inode's reference count hits 0, and the file is removed.

However, if done the opposite and crash occurs, we can't remove its data bitmaps, and data leak would result.

### Notes

- I really need to know what happens

- 3) The primary motivations behind log-structured file system are as follows
  1. Decrease a large number of intermediary processes during a write in FFS (e.g creation of data block, inode)
  2. Utilize the ever increasing capacity of memory, and write a large block of information at once (to boost performance)
  3. To compensate for the slow development of cheap and fast motor for ever increasing capacity and the need for higher data transfer rate

The update process for log-structured file system works by storing write information to in-memory space called **segment**, and write all once its at capacity.

On the other hand, the writing process of FFS works by traveling to target directory, create a inode bitmap, create file inode, update the current directory inode, create a data block in directory inode that points to the file inode, allocate data inode, and write data block for file, and update the current file meta data.

And this data block writing process continues until all are filled.

## Notes

### • Log Structured File System

- Wait. This sounds very similar to **extent-based file system**
- Buffers all updates (including metadata) in an in-memory **segment**, and when segment is full, it is written to disk in one long, sequential transfer to unused part of the disk
- Instead of overwriting files, always writes unused portion of the disk, and reclaim the old space through cleaning
- Motivations
  1. **System memories are growing**
    - \* Data is cached in memory
    - \* Reads are serviced by cache
    - \* Disk traffic is increasingly consists of writes
    - \* File performance  $\approx$  write performance
  2. **There is a large gap between random I/O performance and sequential I/O performance**
    - \* More bits stored on hard drive  $\Rightarrow$  bandwidth of accessing bits  $\uparrow$
    - \* Harder to create cheap, small motors that spin platters faster, and move arm more quickly
  3. **Existing file systems perform poorly on many common workloads**
    - \* Many intermediary writes performed per data block (e.g. Bitmap, inode, data block)
    - \* Many short seeks + rotation delays = performance less than the peak
  4. **File systems are not raid aware**
- How it works (Writing to Disk)

Basic idea: Write all updates (e.g. data blocks inodes) to the disk sequentially (**write buffering**)

1. Buffer updates in an in-memory **segment**
  2. Write the **segment** all at once sequentially when received sufficient number of updates
- Advantages
    1. Has very high performance

- Disadvantages
  1. Is complex
  2. Generates lots of garbages
  3. Scattered old data. Needs to run **compaction** periodically <sup>[2]</sup>
- **Fast File System**
  - Divides inode tables into chunks and stores in different cylinder groups
  - Advantages
    - \* No external fragmentation
  - Disadvantages
    - \* Extra overhead: creates and updates many intermediary files (inode, data block) during a write

### References:

- 1) Ousterhout J. (1991). *The Design and Implementation of a Log-Structured File System*. [link](#)
- 2) Cornell University, Log-Structured File Systems, [link](#)
- 4) The challenge of locating data and metadata on LFS is that both data block and inodes are not at the fixed position.

To compensate for this, LFS has imap that contains the pointers to (i.e. address of) the inode with the latest information, but imap is also placed along side inode and data block.

To compensate for this, LFS has checkpoint region at fixed location that contains the pointers to (i.e. address of) the imap with the latest information.

So when LFS loads, it first reads the checkpoint regions, upload all imap to in-memory, and when file system gives target inode number, it finds inode and its data blocks from there much like UNIX file system.

### Notes

- **Log Structured File System (cont')**
  - Data Block
    - \* Is found by looking for **inode** that point to it



- Inode
  - \* Stores metadata of file
  - \* Is the same as UNIX file system inode
    - Has indirect pointers
    - Has direct pointers
    - Has double indirect pointers
  - \* Is found through a data structure called **inode map**
  - \* Are scattered throughout the disk & keeps moving after update



- Inode Map
  - \* Is a data structure
  - \* Is also moving
    - new data block, inode and imap are placed contiguously
  - \* Locates inode
  - \* Takes inode number as input
  - \* Produces the disk address of the most recent version of the inode
- Checkpoint Region
  - \* Is fixed
  - \* Locates imap
  - \* Contains pointers to (i.e. address of) the latest pieces of inode map
  - \* Is updated periodically (e.g every 30 seconds)
    - Done to ill-affect performance
- How it works (Reading the Disk)
  1. Read the checkpoint region
  2. Read the entire inode map and cache it in memory
  3. Locate inode given inode number of file
  4. Proceed the same as typical UNIX file system

5. a) I am not too sure, but I can deduce as follows:

A crash occurred while deleting directory `bart`.

`file1` inode is removed, but due to crash, inode bitmap remained

Unless intervened, the block may never be available.

### Notes

- **Crash Consistency Problem**

- Desired: **atomic** updates. That is, on crash, the file on write is either in (state 1 - before the file got updated) or (state 2 - after the file got updated)
- Reality: This is not possible
- Is the reason why computers have 'Don't turn off computer' message

- **File System Checker**

- Is implemented in early file system
  - \* Basic Idea: Let inconsistencies happen and fix them later (when rebooting)
- Is used by UNIX tool **fsck** ('file system checker')
- Summary of how it works
  - \* **Inode State**
    - Corruption in file is checked (e.g. does it have valid file type such as directory file, or links)
    - Solved by removing it, and updating the bitmap if inode cannot be fixed easily
  - \* **Inode links**
    - Number of references in each inode is checked
    - Check is done by reading the entire directory tree and building its own link count
    - Solved by fixing the count if there is mismatch, or by moving to `lost+found` directory if there is no directory refers to it
  - \* **Duplicates**
    - Duplicate pointers (i.e. two different inodes pointing to same block) is checked
    - Solved by either removing one of two inodes, or creating a copy for each
  - \* **Bad Blocks**
    - A pointer that points to something outside its partition is checked
    - Solved by removing the block
  - \* **Directory Checks**
    - Making sure that `.` and `..` are first entry is checked
    - Allocation of inodes referred to in a directory entry is checked
    - Making sure that no directory is linked more than once is checked
- Disadvantage



- \* Way too slow. May take Hours.
  - \* Wasteful (Make mistake once, and check everything)
  - \* Doesn't solve all problems (e.g. inode with incorrect data blocks)
- b) When online, user would go around, and work on tasks such as downloading music, coding, or writing documents using Microsoft Word, it's possible that one of the blocks from crash are already in use by others.

Because `fsck` corrects bad blocks, duplicates, inconsistencies in inode metadata, and directories, it's possible that on correction, the normal files and directories may become corrupted. (not good)

- c) The full data journaling is performed in following steps
1. **Journal Write:** Write contents of the transaction (containing TxB and contents of the update) to the log; wait for the writes to complete
  2. **Journal Commit** Write transaction commit block (containing TxE) to the log; wait for the write to complete
  3. **Checkpoint:** Write contents of the update to their final location in the file system
  4. Mark the transaction free in the journal by updating **journal superblock**

The metadata journaling is performed in followings steps:

1. **Data Write:** Write data to final location; wait for completion
2. **Journal Metadata Write:** Write the begin block and metadata to the log; wait for writes to complete
3. **Journal Commit:** Write the transaction commit block to the log; wait for the transaction to complete
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system
5. **Free:** Mark the transaction free in journal superblock

From the two, we can see that unlike data journaling that writes bitmap, inode and data block to journal before writing to the file system, metadata journaling writes data block to file system first before bitmap and inode are written to journal of which later written to the file system.

It is this step, where data block isn't loaded to filesystem along with bitmap and inode, that metadata journaling doesn't provide full consistency.

### Notes

**Question** What does full consistency mean in journaling?

- After step 4 of data journaling, where are the next transaction begin and end placed?
- **Journaling**
  - Is a popular solution to **crash-consistency problem**
  - Many file systems use this idea (e.g. ext3, ext4, windows NTFS)
  - Basic idea
    - \* before overwriting the structures in place, write down (in a well-known location) a little note of what you are about to do
    - \* If crash occurs, read note and try again

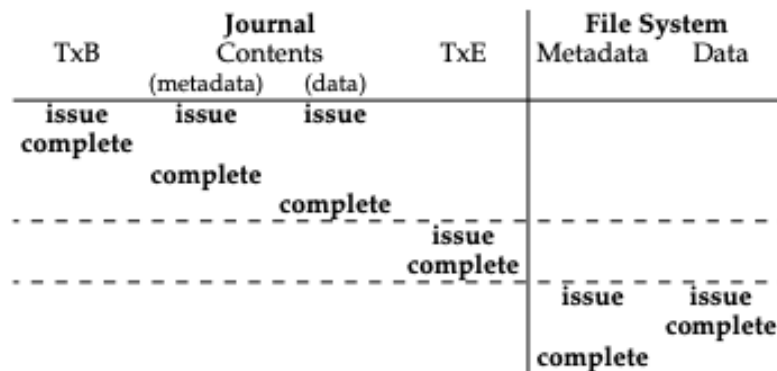


- Advantage
  - \* Greatly reduces amount of work required during recovery
- **Transaction Beginning (TxB)**
  - Where does computer read update instruction (journal ? journal superblock ?)?
  - In data Journaling, where is committed data generated and stored prior to putting it in file system?
  - Includes information about current update
  - Contains **Transaction Identifier** or TID
- **Transaction End (TxE)**
  - Is marker of the end of transaction
  - Also contains **Transaction Identifier** or TID
- **Checkpointing**
  - Act of overwriting of old structure in the file system between **transaction beginning** and **transaction end**
- **Journaling Superblock**
  - Records information on which transactions have not yet been checkpointed
  - Oldest and newest non-checkpointed transactions exist here
  - Is different from file system superblock
- **Data Journaling**

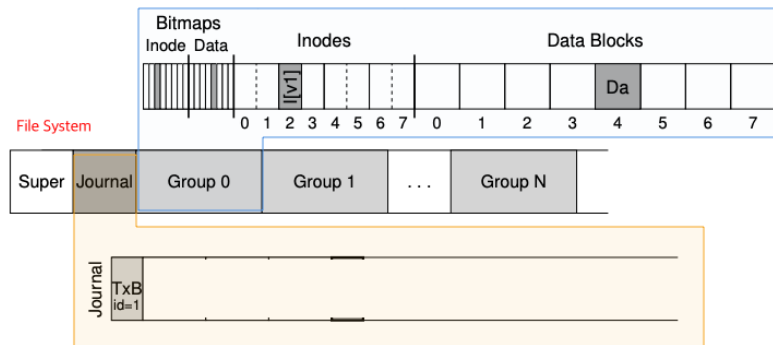


**Important** Is written to journal before putting onto file system!!!

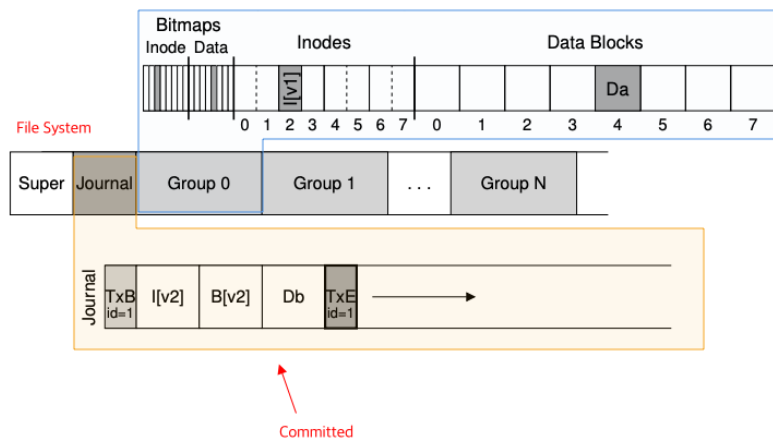
– Steps



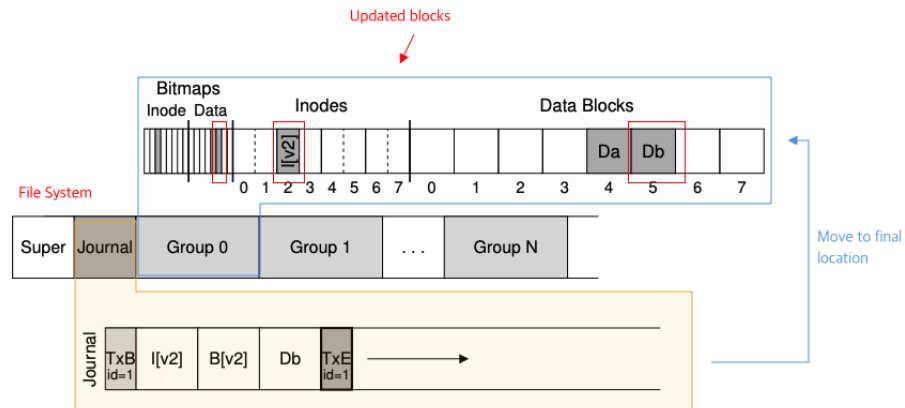
1. **Journal Write:** Write the contents of the transaction (including TxB, metadata and data) to log



2. **Journal Commit:** Write the transaction commit block (containing TxE) to log; wait wait for write to complete  
 \* After this, transaction is **committed**



3. **Checkpoint:** Write the contents of the update (metadata and data) to their final on-disk location



4. **Free:** Mark the transaction free in the journal by updating the journal superblock



5. Repeat until done

- Disadvantage
  - \* Each data block is written twice
- Recovery Steps
  - \* Crash at step 1 → skip pending update
  - \* Crash during step 2 and 3 → replay the update
    - Happens during boot

#### • Metadata Journaling

- Goal: Reduce number of writes
- Data block is written to file system first
- Metadata (inode and bitmap information) are written to journal before checkpoint
- Is order dependent
  - \* e.g. I[v2] and B[v2] make to disk and data block does not
  - \* If data block is a garbage data, file-system will assume all is okay
  - \* Writing data block first guarantees that a pointer will never point to garbage



– Steps



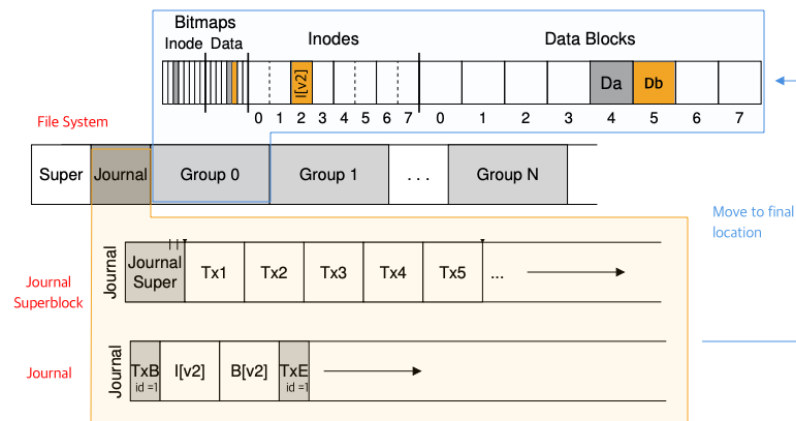
1. **Data Write:** Write data to final location; wait for completion



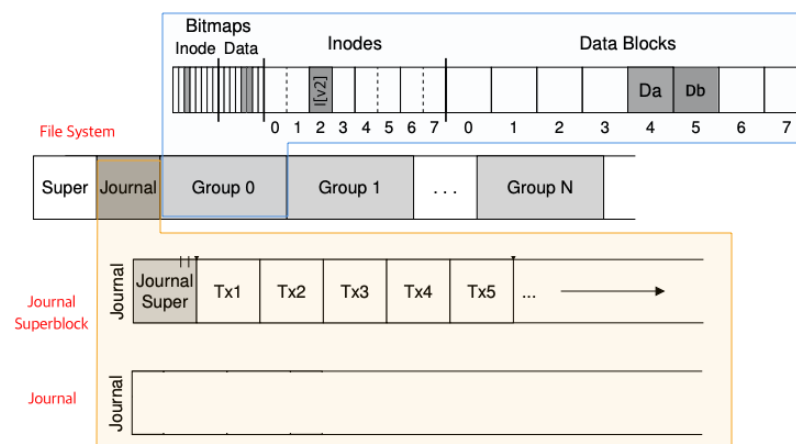
2. **Journal Metadata Write:** Write the begin block and metadata to the log; wait for writes to complete
3. **Journal Commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete



4. **Checkpoint Metadata:** Write the contents of the metadata update to their final locations within the file system



5. **Free:** Mark the transaction free in journal superblock



- Block Reuse
  - \* Never reuse blocks until checkpointed out of the journal
- Advantage
  - \* Solves double write problem in **data journaling**

d) Question omitted. Is not covered in readings

### Notes

- **Flash-Based SSDs**
  - Is based on NAND-based flash
  - Retains information despite powerloss
- **Block**
  - Is the largest unit
  - Are typically of size 128 KB or 256 KB
- **Page**
  - Is unit smaller than Banks
  - Is the smallest unit of data for memory management in a virtual memory<sup>[1]</sup>
  - Are few KB in size (e.g 4KB)
- **Bits**
  - Is the smallest unit
  - Is unit smaller than planes

### References

1)

6. a) Algorithm A completes faster on Average. Reason is that algorithm A (moving the data blocks in A and freeing the existing) has fewer number of updates than algorithm B (swapping the data blocks, and dealing with remaining data blocks).

b) **Correct Solution**

My best attempt is that the Algorithm B leads to better performance because disk arm doesn't have to move as much. (moving from top to bottom)

c) The changes made to inode are:

- Location of data blocks, i-number
- time information

Location of data blocks, i-number because inode and its' data block's location in hard drive are being changed.

Time information (Time modified), because its pointers pointing to data block are being modified.

### **Correct Solution**

The changes made to inode are:

- Location of data blocks, i-number

Because inode and its' data block's location in hard drive are changed.

- Inode bitmap, data bitmap

Because bitmap contains information about where inode and data blocks are occupied, and this information has changed

- Inode metadata - time information

Because the pointers and data blocks of an inode are accessed and are modified

d) The changes that would need to be made to directory metadata are

- i-number in directory entry
- Time information

i-number in directory entry because files and child directories inside the current directory are being moved around.

Time information (Time modified) because i-number in directory entries are being modified.

e) The changes made are

- Location of each data blocks in linked list

Because they are relocated during the defragmentation process.

- The value of next member variable in each data block

Because the block location have change during the defragmentation process.

- metadata of inode - time information, i-number, starting block i-number, final block i-number

Because inode information must be modified to reflect changes made to data blocks.

### Correct Solution

The changes made are

- Location of each data blocks in linked list
- The value of next member variable in each data block
- Starting block i-number, and final block i-number of each file and directory in file system

The reason is the data blocks have relocated during the defragmentation process.

f) Omitted, because it's currently not covered (it's not on the reading list).

### Notes

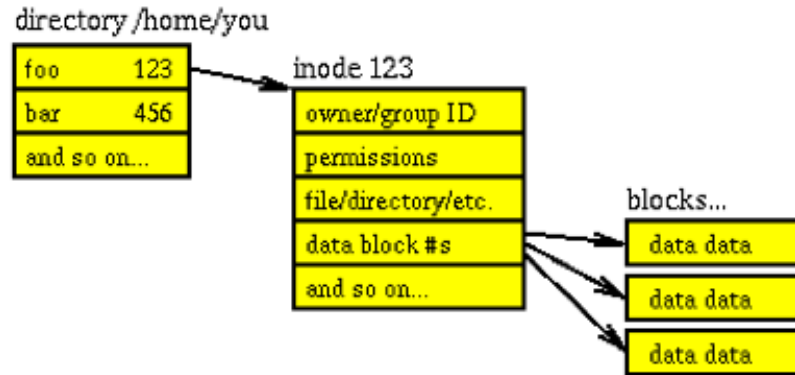
**Question** In question 4.c, does the inode in phrase 'what changes are made to inode' referring to inode's metadata?



**Question** Is time information changed when defragmentation occurs?

- **Directory**

- Directory structure looks as follows



- **Linked List Allocation**

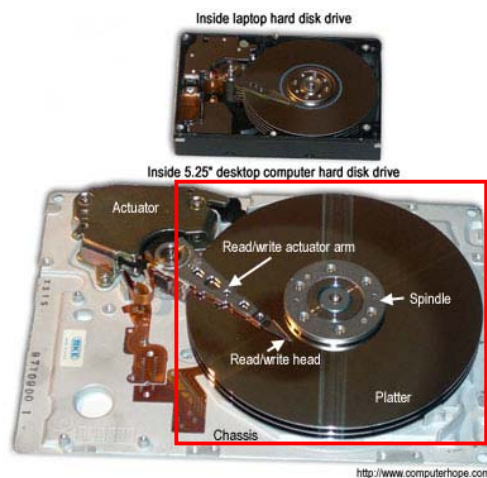
- Does not have inode (i need a strong verification on this)

- **Hard Disk Drive**

- Is a non-volatile data storage device

- **Platter**

- Is a circular hard surface on which data is stored
- One or more aluminum, glass, or ceramic disk that is coated in a magnetic media [1]
- All modern drives use glass or glass-ceramic platters [2]



- **Spindle**

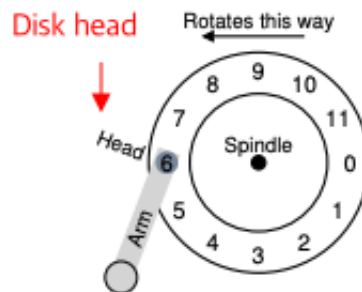
- Is a motor that spins the platter around

- **Track:**

- is a data storage ring on a computer hard drive that is capable of storing information.



- **Disk Head**



- Is where process of reading and writing is accomplished

- **Disk Arm**

–

7. a) All of the above

- a - inode bitmap
- b - data bitmap
- c - file inode
- d - directory inode
- e - directory data block

b) f - Multiple of the problems above

because of

- c - Inode leak (inode block is lost for any future use)

**Correct Solution**

c - Inode leak

c) a - No inconsistency

d) f - Multiple of the problems above

because of

d - Multiple file paths may point to same inode

e - Something points to garbage

### Notes

- Realized that 'Multiple file paths may point to same inode' occurs when only directory inode is updated

e) a - No Inconsistency

### Correct Solution

c - Inode leak

f) f - Multiple of the problems listed above

because of

c - Inode leak

d - Multiple file paths may point to the same inode

e - Something points to garbage

g) a - No Inconsistency

8. 1) **s1**

Since `mygrade-copy2.txt` is a soft link to `mygrade-copy.txt`, the removal of `mygrade-copy.txt` would result in `mygrade-copy2.txt` becoming a dangling reference.

**s2**

There is no problem in this case. The symbolic link to file `mygrades.txt` is removed, but the hardlinks to the inode of file `mygrades.txt` remains.

The total number of references to inode of file `mygrades.txt` is 2.

**s1**

There is no problem at all. The `mygrade-copy2.txt` is a soft link to file `mygrade-copy.txt` of which points to the inode of file `mygrade.txt`

However, because the file `mygrade.txt` is removed, the number of references in the inode of `mygrade.txt` would reduce from 2 to 1.

**s2**

There is no problem in this case. `mygrade-copy2.txt` is a hardlink pointing to the inode of file `mygrade.txt`.

So, when removed, `mygrade-copy2.txt` still points to the inode of `mygrade.txt`.

However, in inode of file `mygrade.txt`, the total number of reference count would reduce from 2 to 1.

- 2) The number of reference count to inode of the file is reduced to 0.

FS frees the file's inode, and its data blocks by deallocating them in bitmap.

FS updates its parent directory, freeing the directory entry in second data block for it to be used by others.

- 3) Check is done by going through each pointers in files and directories to make sure that the data blocks match the information in data bitmap.

If there is mismatch, then the bad block is removed.

- 4) Omitted for now. Topic not in reading list.

9. a) The largest file that can be represented by an inode is:

$$(12 + 1024 + 1024^2) * 4KB \approx 4.20GB.$$

- b) In total,

$12 + (2 \times 1024) + (3 \times 1024^2) \approx 3.15$  million accesses are required to read the file of maximum size in sequential order.

- c) The two ways of implementing directories are

- Using lists
  - Each directory entry is of format
 
$$(\text{"User-readable-file-name"}, \text{i-number})$$
  - Is easy to implement
  - Suffers in performance. (we have to check each individual entry to find the correct file)
- Using hasmap
  - key of hasmap is user-readable file name, and value is i-number of file
  - Takes space
  - Excellent in performance.

10. a) The largest possible file is

$$(12 + 1024 + 1024^2 + 1024^3) * 4KB \approx 4.30TB.$$

b) A size of block is 4KB.

So,  $(12 + 1024 + 1024^2 + 1024^3) = 1,074,791,436$  blocks are need to create this file.

c) • **Symbolic link**

- Can create link across partitions
- Can create link to a directory
- File system needs to follow the link
- Removing the linked file would result in dangling link/reference

• **Hard link**

- Cannot create a link to a directory
- Cannot create a link across partitions
- No extra work is involved in following the link
- Removing hardlink removes file only when its hardlink count reaches 0.