1 Exam Related Questions and Tips

- What is hard link? What is soft link? What are the differences between the two?
- 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

2 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 readible 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

File can be read by owner

File can also be written by owner

File can also be read by group

File can also be read by others

```
#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);
...
```

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()?)

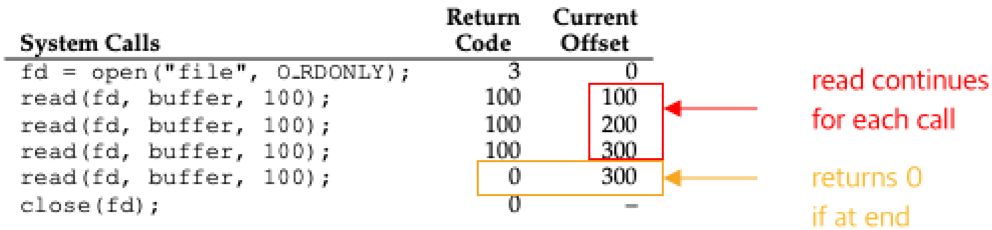
- (read) (read file)
  - Is a system call
  - **Syntax:**

    ```
    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

  <u>**Example**</u>

```
char buf[4096];
int fd = open("/a/b/c", 0); // 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
fd = open("file", O_RDONLY);	3	0
read(fd, buffer, 100);	100	100
read(fd, buffer, 100);	100	200
read(fd, buffer, 100);	100	300
read(fd, buffer, 100);	0	300
close(fd);	0	–

- `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

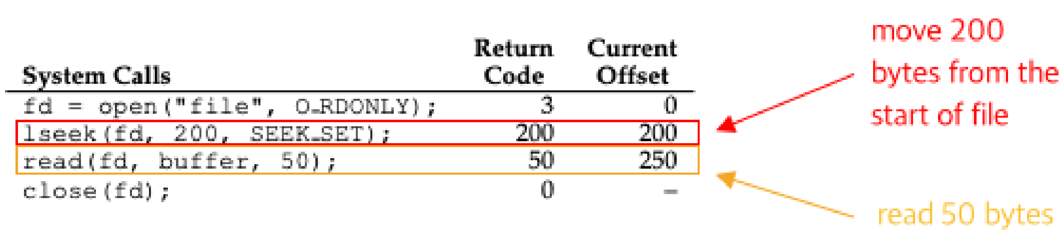
- **Syntax:**

  `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
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	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");
```

e.g. "hello\n"

- `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

```c
#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%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 <u>the first entry</u> 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

  <u>Example</u>

```
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

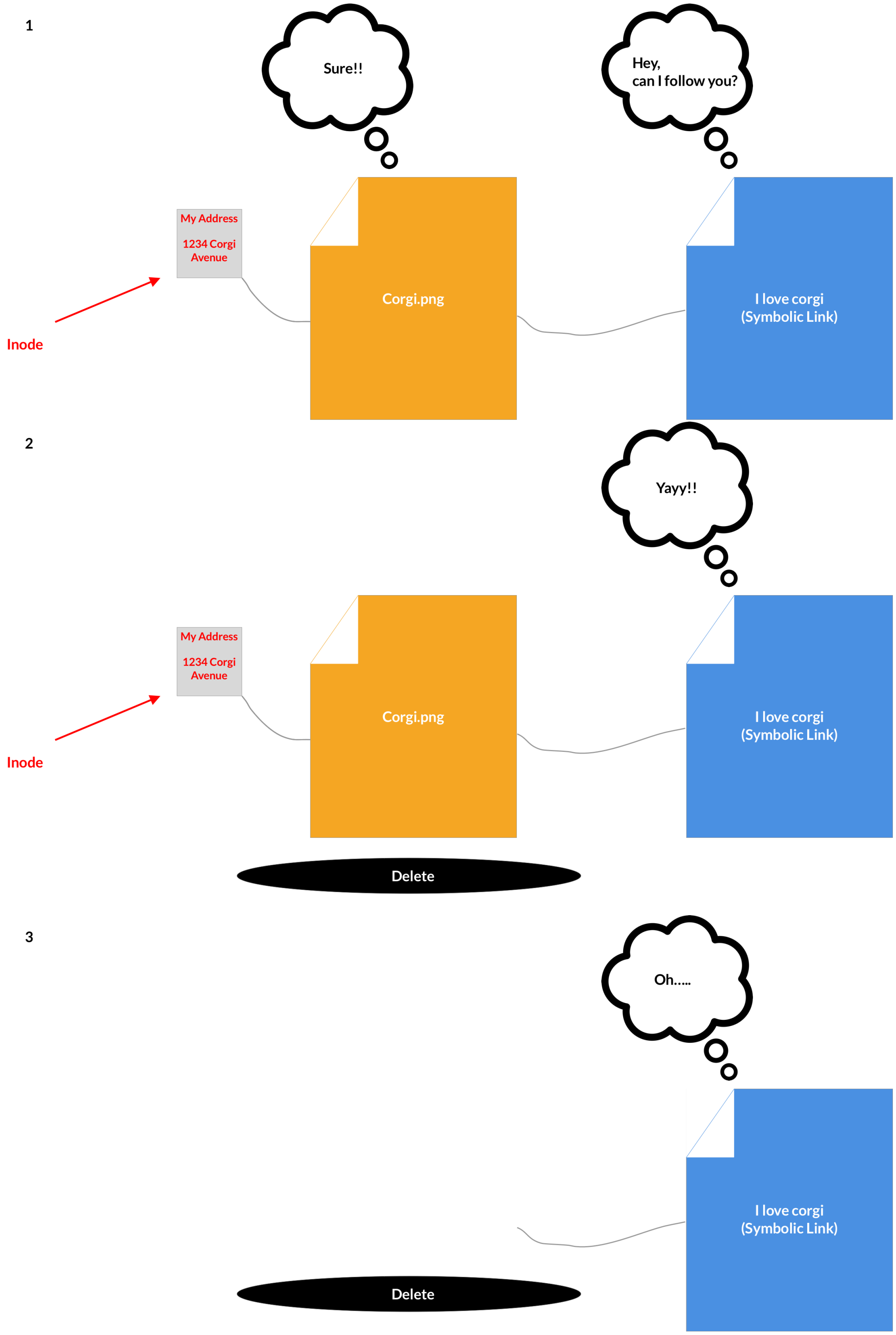
```
#include <unistd.h>

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
```

3 Symbolic Link:

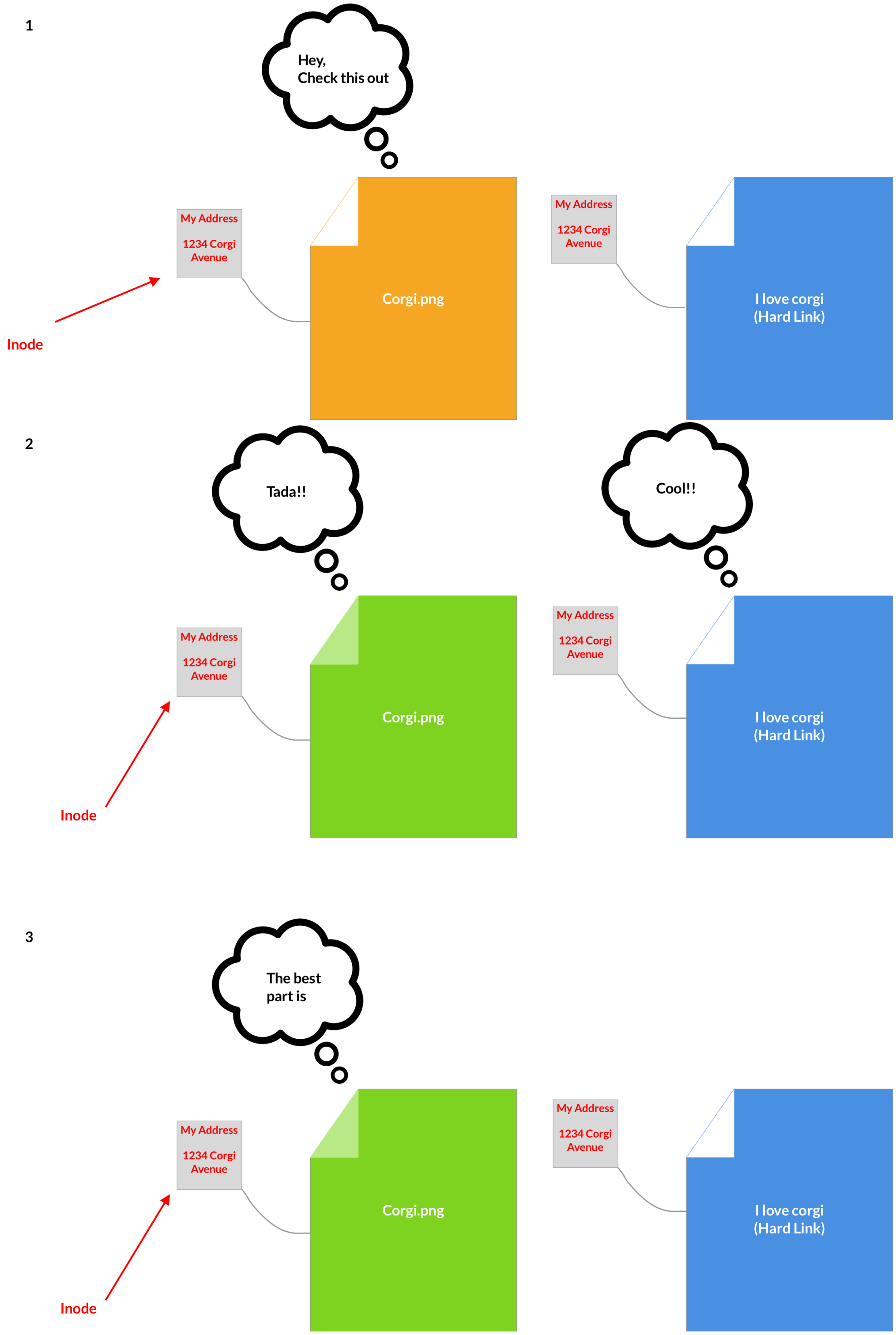
- Is directory entry containing "true" path to the file
- Is a shortcut that reference to a file instead of inode value [2]

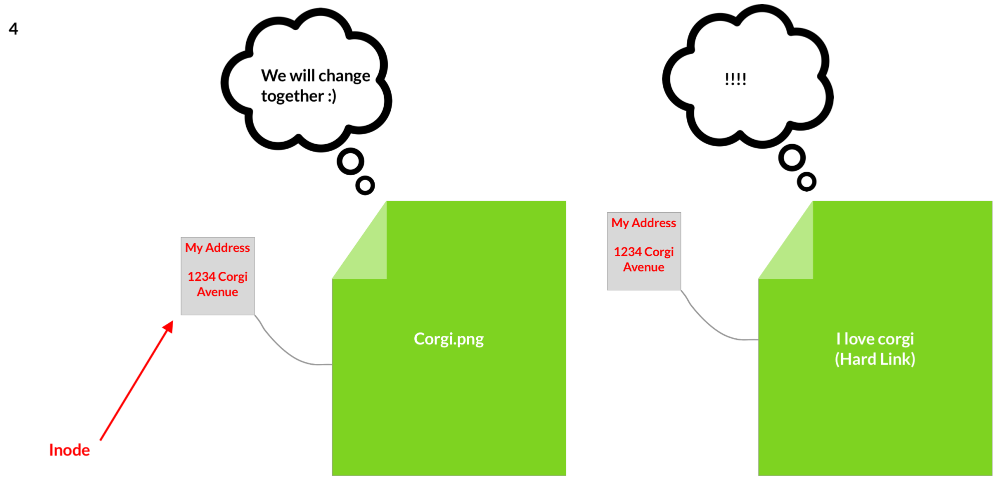


4 Hard Link:

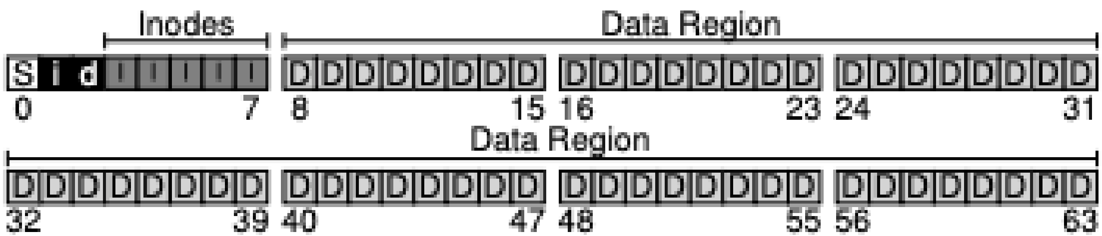
- Is a direct reference to a file via its inode [2]

- Is second directory entry identical to first





5 Index-based File System



- Has following parts
  - Superblock
  - Inode Bitmap
  - Data Bitmap
  - Inodes
  - Data Region
- Each block in file system is 4KB
- Uses a large amount of metadata per file (especially for large files)

6 Kilobyte

- 1 kilobyte is 1024 bytes

7 File*

- is an array of bytes which can be created, read, written and deleted
- low-level name is called **inode number** or **i-number**

8 Static Partitioning

- Divides resources into fixed proportion <u>once</u>
  - 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
  -

9 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 perodic frozen feeling when loading screen)

10 External Fragmentation

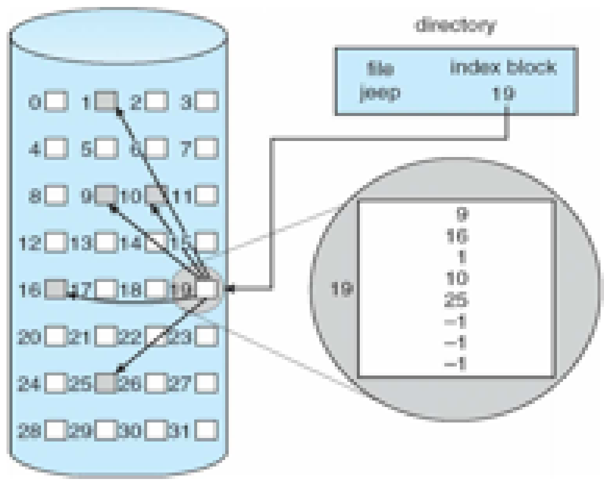
- Is various free holes that are generated in either your memory or disk space. [8]
- Are available for allocation, but may be too small to be of any use [8]

11 Internal Fragmentation

- Is wasted space within each allocated block [8]
- Occurs when more computer memory is allocated than is needed

12 Disk Layout Strategies

12.1 Index-Based Allocation



- File metadata stored in inode
- Has 15 blocks of pointers
  - First 12 are direct block pointers
  - 13th is a single indirect block pointer
    - Address of block containing address of data blocks
  - 14th is a double indirect block pointer
    - Address of block containing address of single indirect blocks
  - 15th is a triple indirect block pointer
    - Address of block containing address of double indirect blocks
- Index block contains pointers to many other blocks
- Advantages
  - No external fragmentation
  - Handles random access better

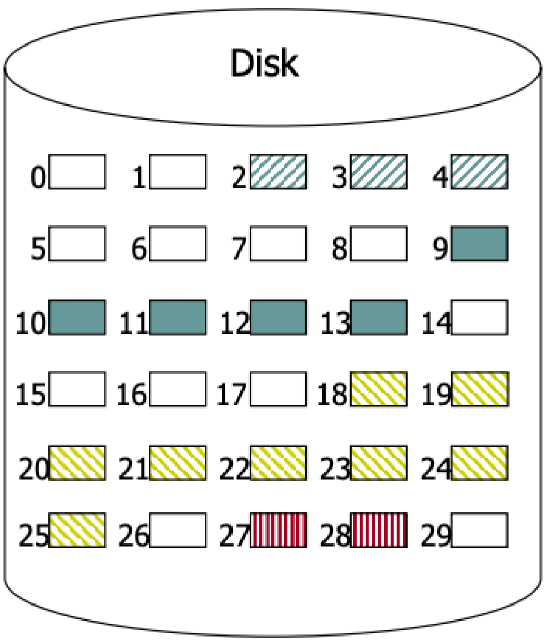
  - Files can be easily grown

- Disadvantage
  - May reuiqre multiple, linked index blocks

Example

Linux's ext2, ext3

12.2 Contiguous-Based Allocation



directory

File Name	Start Blk	Length
File A	2	3
File B	9	5
File C	18	8
File D	27	2

- **Inode** stores starting block and total length
- Is simply a disk pointer plus a length (in blocks)
  - Together, is called **extent**
- Often allows more than one extent
  - resolve problem of finding continuous free blocks
- Inode stores starting block and total length
- Is less flexible but more compact

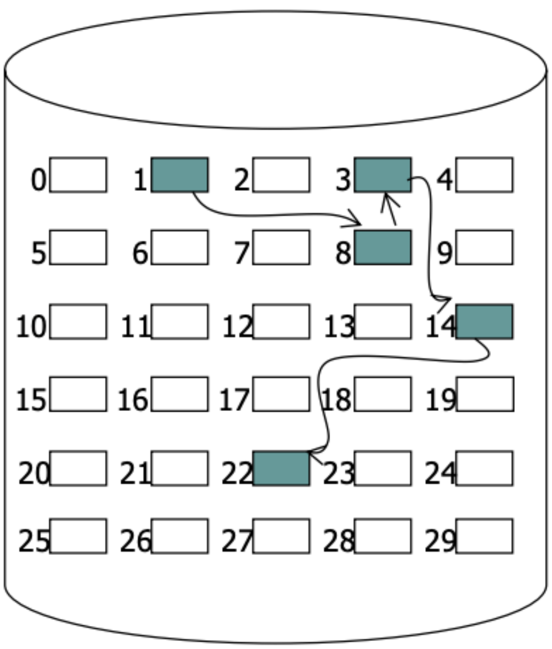
- Works well when there is enough free space on the disk and files can be laid out contiguously

  Example

  Linux's ext 4

- Advantage
  - Is simple
    - Finding data block = beginning of data block + length
  - Fast, simplifies directory access and allows indexing
- Disadvantage
  - Growing file size could cause problems
  - Inflexible, causes **external fragmentation**
  - Requires compaction

12.3 Linked Allocation



directory

File Name	Start Blk	Last Blk
...	...	...
File B	1	22
...	...	...

- **Inode** stores the starting block and last block in the directory
- Each block in file contains a pointer to the next block
- Disadvantage

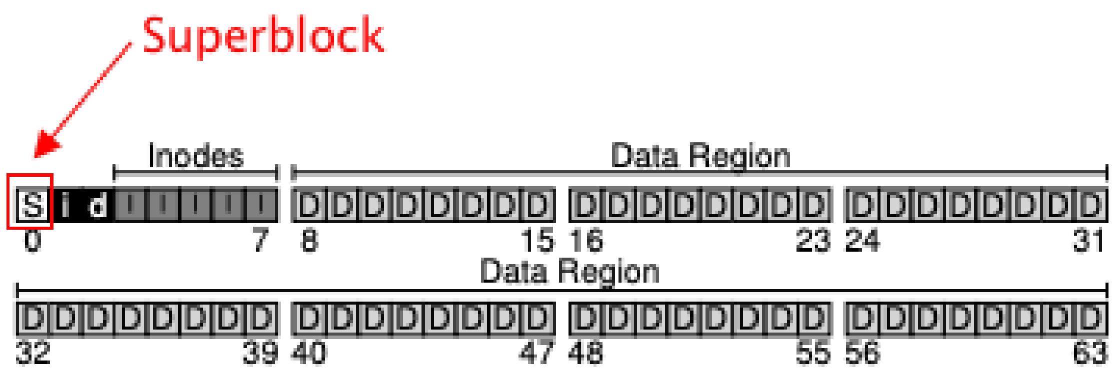
- If one of the data becomes corrupted, we lost pointers to rest of file
- O(n) to find/read nth block (Takes really long time)

Example

Windows' FAT file system

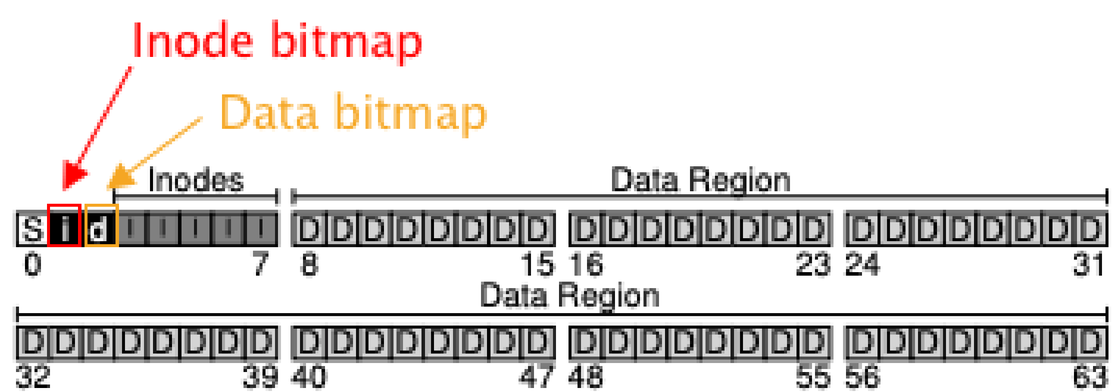
13 File System Implementation

14 Superblock



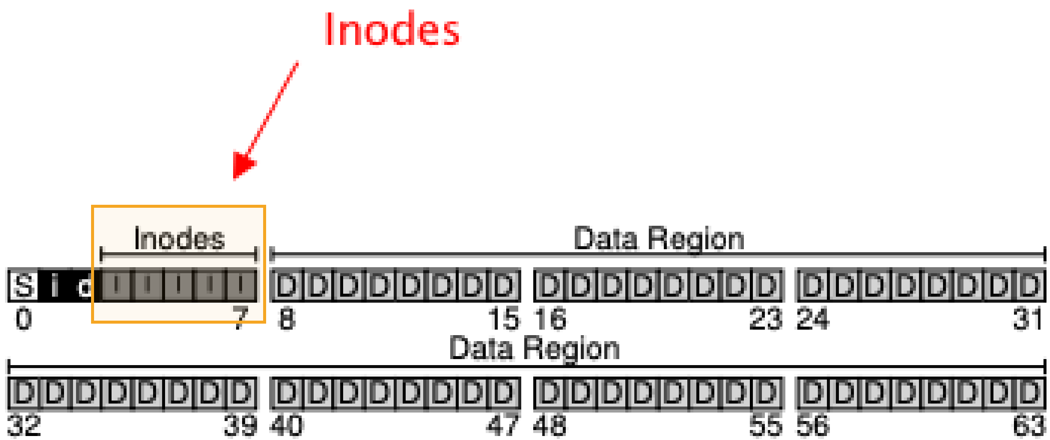
- Contains information about the following
  - The number of inodes and data blocks in a particular file system
  - The magic number of some knd to identify the file system type
  - Where the inode table begins
- Is read first on mount before attaching to file system

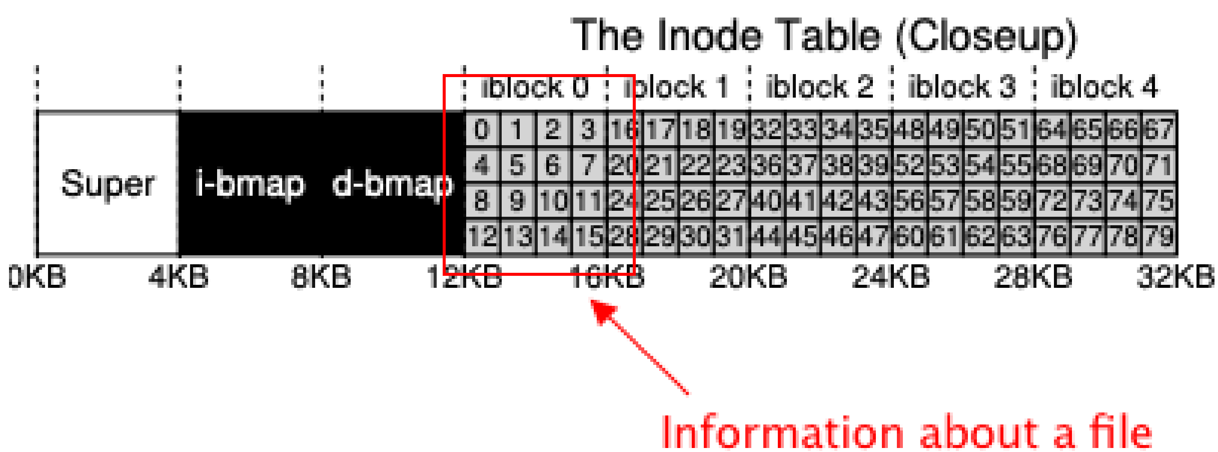
14.1 Inode/Data bitmap



- 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

14.2 Inode

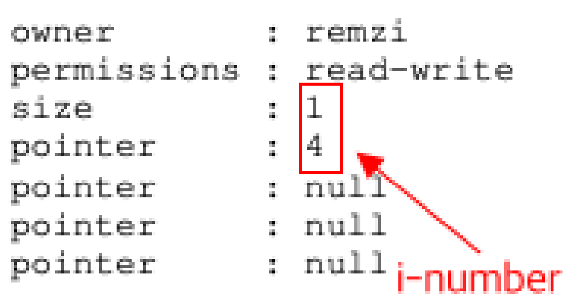




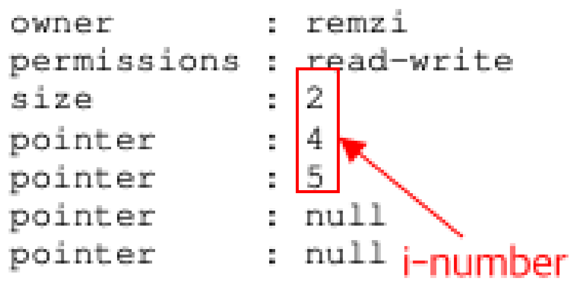
- Is a short form for **index node**
- Contains all the information you need about a file (i.e. metadata)
  - File Type
    - e.g. regular file, directory, etc
  - Size

- Number of blocks allocated to it
- Protection information
  - such as who owns the file, as well as who can access it
- Time information
  - e.g. When file was created, modified, or last accessed
- Location of data blocks reside on disk

- total size may vary
- inode pointer has size of 4 byte
- Has 12 **direct pointers** to 4KB data blocks
- Has 1 **indirect pointer** [when file grows large enough]
- Has 1 **double indirect Pointer** [when file grows large enough]
- Has 1 **triple indirect Pointer** [when file grows large enough]
- Inode before update



- Inode after update



15 Data Block

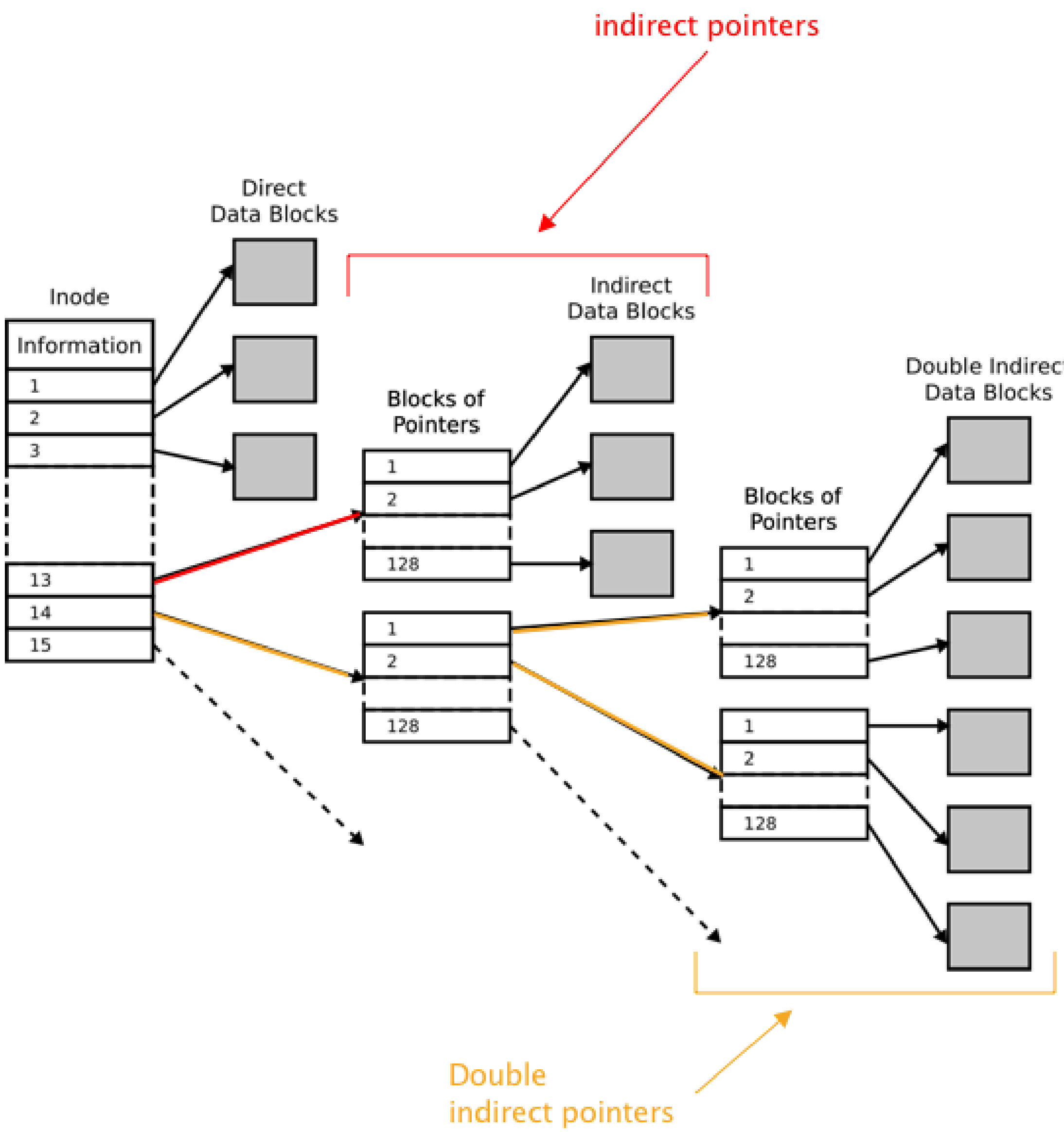
- Size of each block is 4KB

15.1 Indirect Pointers

- Is allocated to data-block if file grows large enough
- Has total size of 4 KB or 4096 bytes
- Has $4096/4 = 1024$ pointers
- Each pointer points to 4KB data-block
- File can grow to be $(12 + 1024) \times 4K = 4144KB$

15.2 Double Indirect Pointers

- is allocated when single indirect pointer is not large enough
- each pointer in first pointer block points to another pointer block
- has 1024^2 pointers
- each of 1024^2 pointers point to 4KB data block
- File can grow to be $(12 + 1024 + 1024^2) \times 4K = 4198448KB$ or $\approx 4.20GB$



15.3 Triple Indirect Pointers

- is allocated when double indirect pointer is not large enough
- has 1024^3 pointers
- each of 1024^3 pointers point to 4KB data block
- File can grow to be $(12 + 1024 + 1024^2 + 1024^3) \times 4K = 4299165744KB$ or $\approx 4.00TB$

15.4 Reading a File from Disk

Example

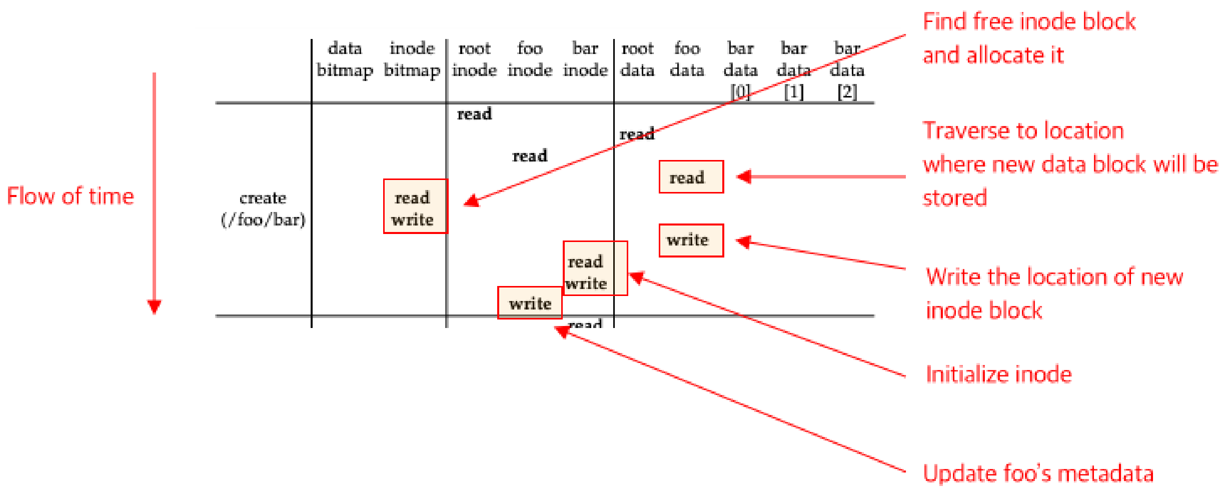
When

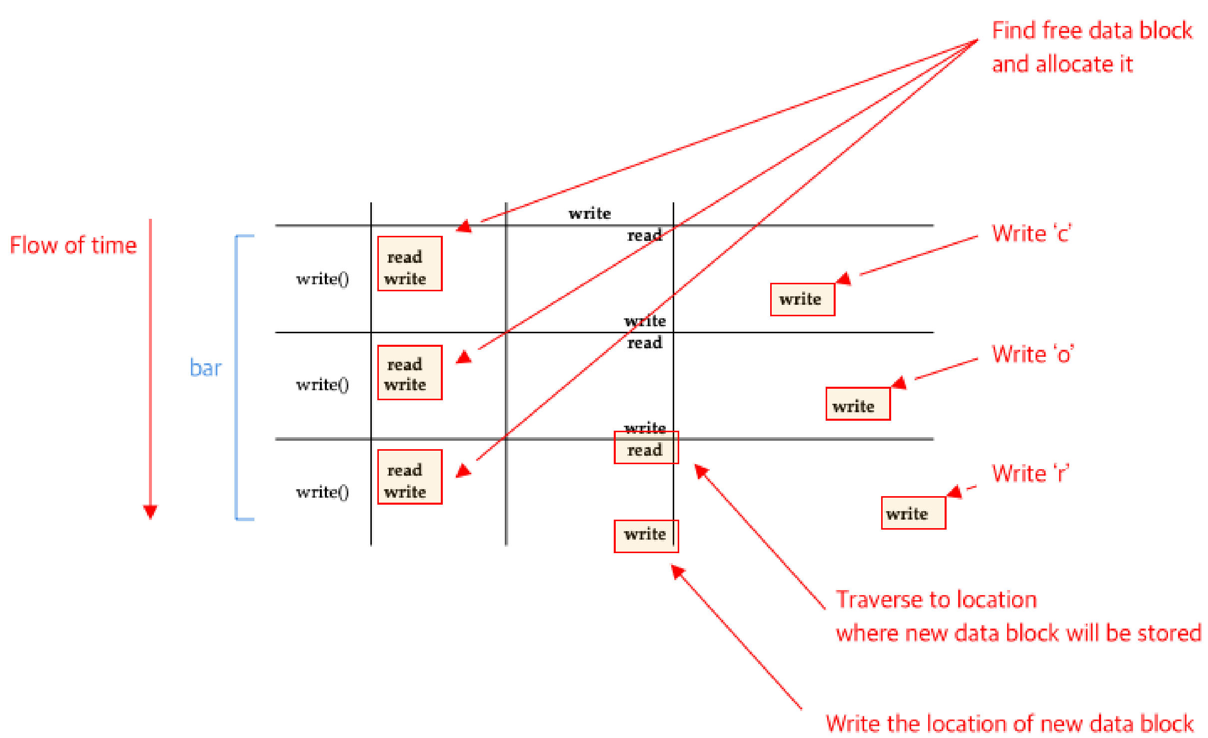
`open("/foo/bar", O_READONLY)`

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. Trasnfer 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.

15.5 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

15.6 Static Partitioning

- Divides resources into fixed proportion <u>once</u>
  - 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
  -

15.7 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**

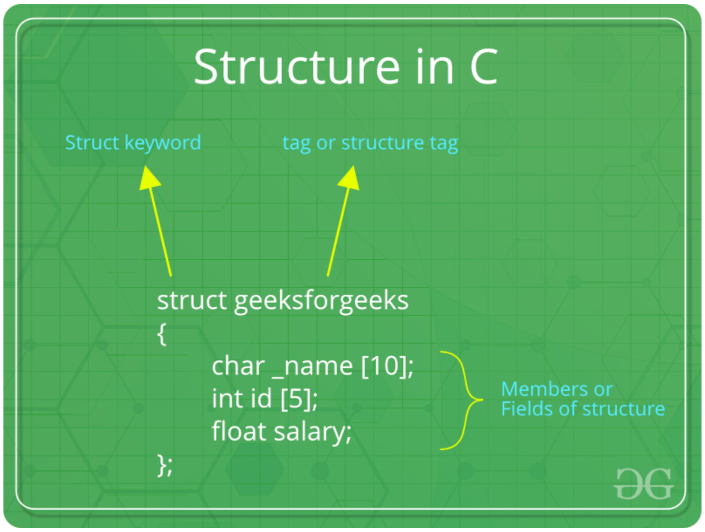
- Disadvantages
  - More complex to implement
  - Could lead to worse performance
    * e.g idle resource got consumed by others and take long time to reclaim it when needed (the perodic frozen feeling when loading screen)

Example

Linux's ext4 file system

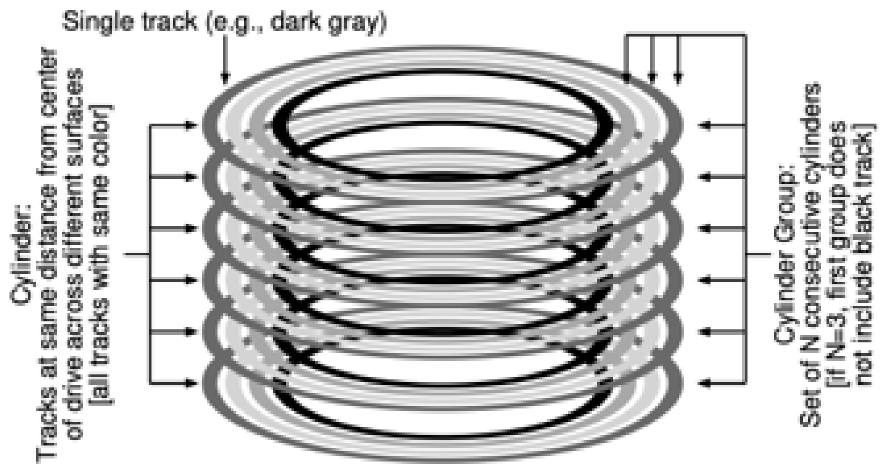
16 Fields

- Is the members in a structure

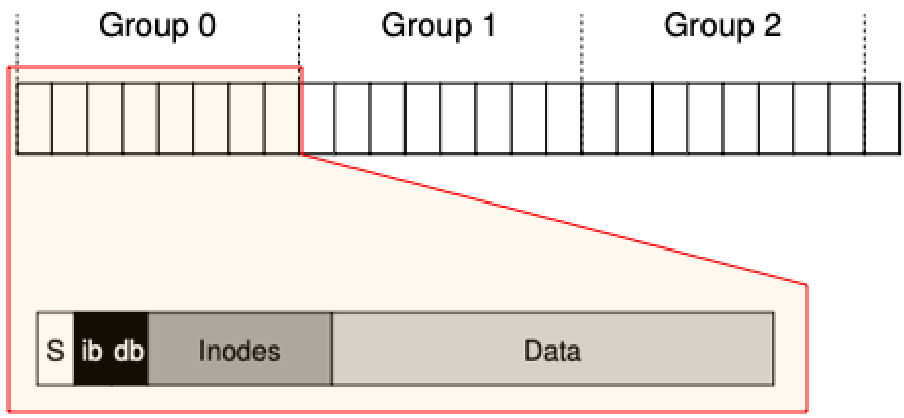


17 Fast File System

- Modern file system has same APIS (`read()`, `write()`, `open()`, `close()`)
- Divides inode/bitmap tables into chunks and stores in different **cylinder groups**



- Each **block group** or **cylinder group** is consecutive portion of disk's address



- Advantages
  - No external fragmentation
- Disadvantages
  - Extra overhead: creates and updates many intermediary files (inode, data block) during a write

17.1 FFS Policies: Allocating Files and Directories

- Basic Idea: keep related stuff together, and keep related stuff far apart
- Directories Step
  1) Find the **cylinder group** with a low number of allocated directories and a high number of free inodes

  - low number of allocated directories → to balance directories across groups
  - high number of free nodes → to subsequently be able to allocate a bunch offiles

2) Put directory data and inode to the **cylinder group**

- Files Step

  1) Allocate the data blocks of a file in the same **cylinder group** as its inode
  2) Place all files in the same directory in the cylinder group of the directory they are in

  Example

  On putting `/a/c`, `/a/d`, `/b/f`, FFS would place

  - `/a/c`, `/a/d` as close as possible in the same **cylinder group**,
  - `/b/f` located far away (in some other **cylinder group**)

18 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 ≈ write performance
  2. **There is a large gap between random I/O performance and sequential I/O performance**
     - More bits stored on hard drive ⇒ bandwith of acessing bits ↑
     - 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

- Advatages

  1. Has very high performance

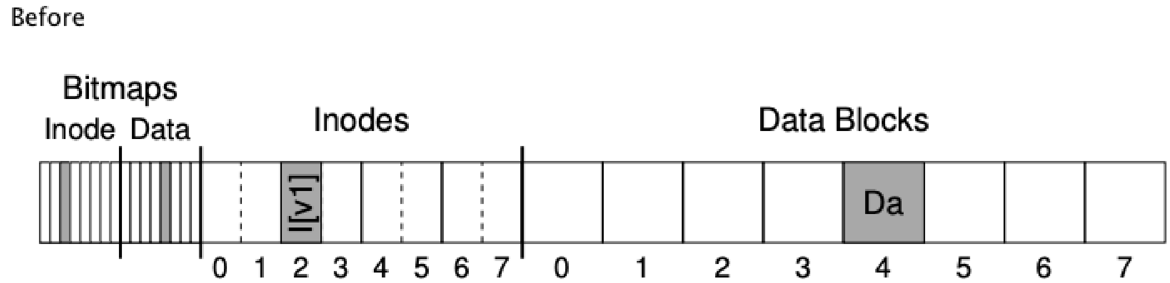
- Disadvantages

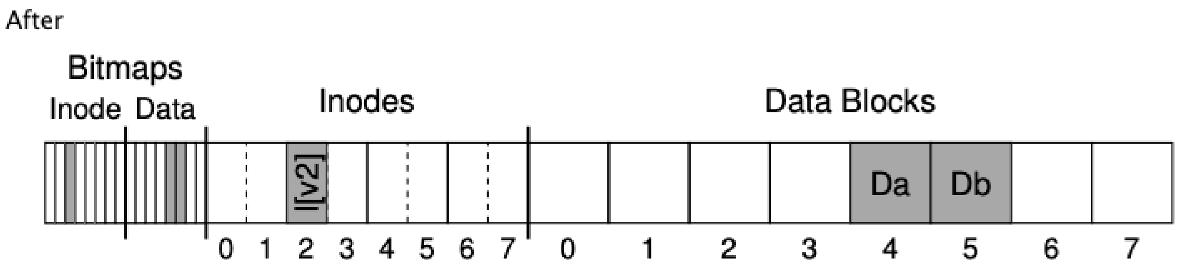
  1. Is complex
  2. Generates lots of garbages
  3. Scattered old data. Needs to run **compaction** periodically [2]

19 Crash Consistency Problem: File System Checker

- 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

19.1 Crash Scenarios





1) Just the data block (Db) is written to disk

- No inode that points to it
- No bitmap that says the block is allocated
- It is as if the write never occured
- There is no problem here. All is well. (In file system's point of view)

2) Just the updated inode (I[v2]) is written to disk

- Inode points to the disk where `Db` is about to be written
- No bitmap that says the block is allocated
- No `Db` is written
- Garbage data will be read
- Also creates **File-system Inconsistency**
  - Caused by on-disk bitmap telling us Db 5 is not allocated, but inode saying it does

3) Just the updated bitmap (B[v2]) is written to disk

- Bitmap indicates tht block 5 is allocated
- No inode exists at block 5
- Creates **file-system inconsistency**
- Creates **space-leak** if left as is
  - block 5 can never be used by the file system

4) Inode (I[v2]) and bitmap (B[v2]) are written to disk, and not data

- File system metadata is completely consistent (in perspective of file system)
- Garbage data will be read

5) Inode (I[v2]) and data block (Db) are written, but not the bit map

- Creates **file-system inconsistency**
- Needs to be resolved before using file system again

6) Bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2])

- Creates **file-system inconsistency** between inode and data bitmap
- Creates **space-leak** if left as is
  - Inode block is lost for future use
- Creates **data-leak** if left as is
  - Data block is lost for future use

19.2 File System Checker

- 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 is 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)

20 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

20.1 Transaction Beginning (TxB)

- Where does computer read update instruction (journal ? journal superblock ?)?
- In data Journaling, where is comitted data generated and stored prior to putting it in file system?
- Includes information about current update
- Contains **Transaction Identifier** or TID

20.2 Transaction End (TxE)

- Is marker of the end of transaction
- Also contains **Transaction Identifier** or TID

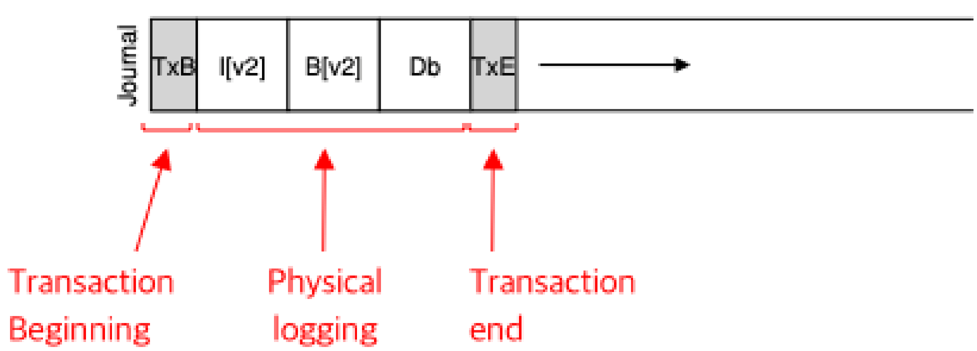
20.3 Checkpointing

- Act of overwriting of old structure in the file system between **transaction beginning** and **transaction end**

20.4 Journaling Superblock

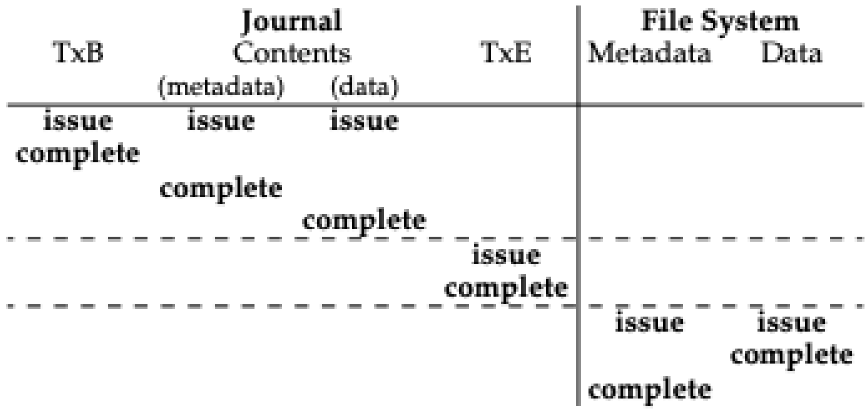
- Records information on which transactions have not yet been checkpointed
- Oldest and newest non-checkpointed transactions exist here
- Is different from file system superblock

20.5 Data Journaling

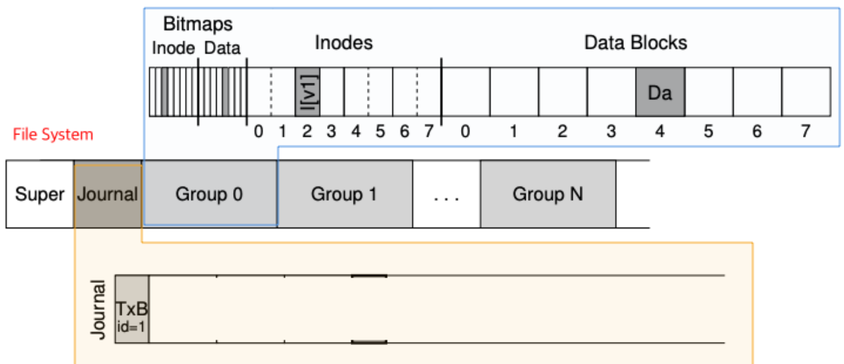


Important Is written to journal before putting onto file system!!!

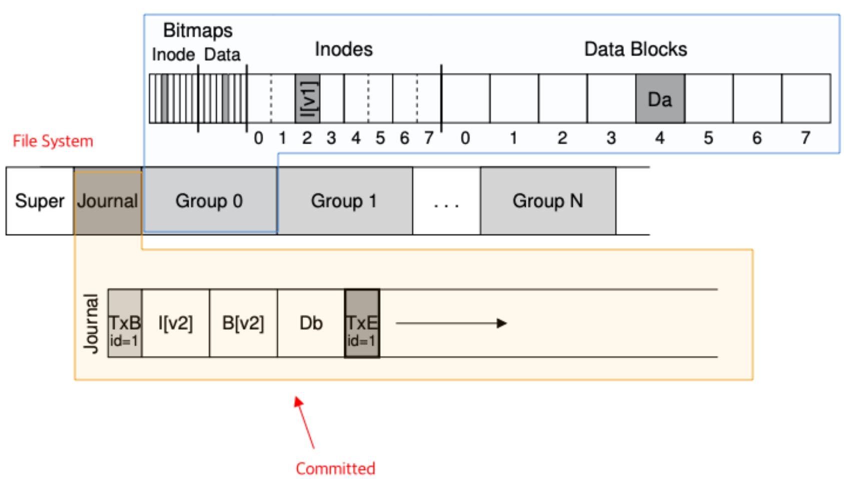
- Steps



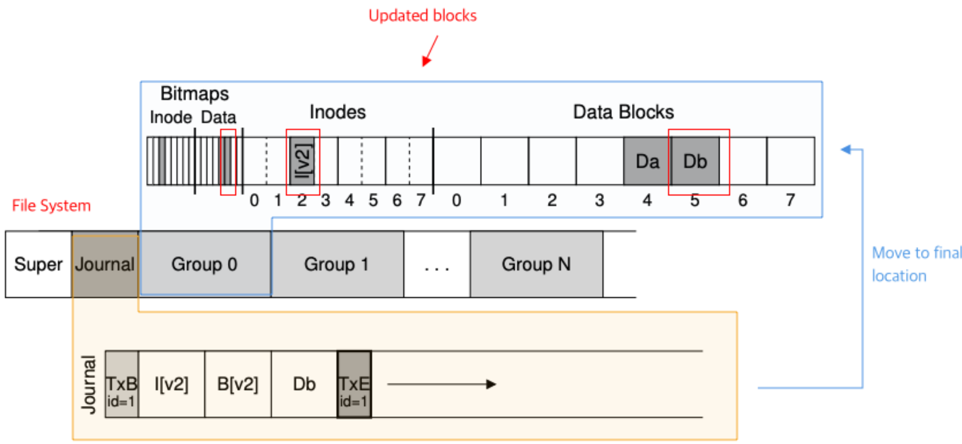
1. **Journal Write**: Write the contents of the transaction (including TxB, metadata and data) to log



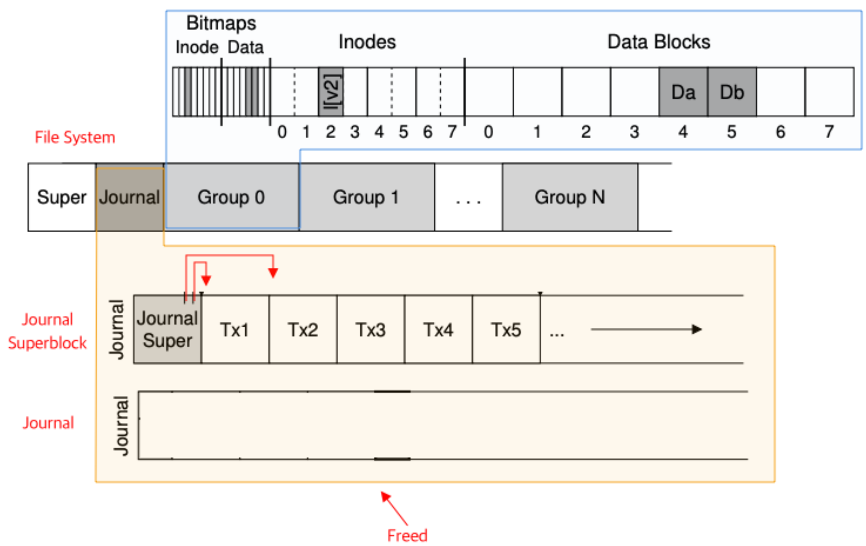
2. **Journal Commit:** Write the transaction commit block (containing TxE) to log; wait wait for write to complete
   - After this, transaction is **committed**



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

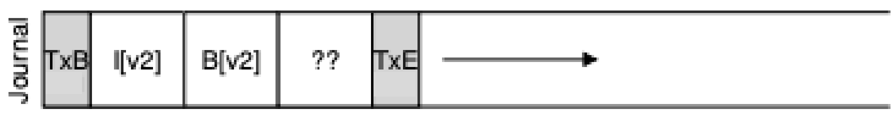


5. Repeat until done

- 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

20.6 Metadata Journaling

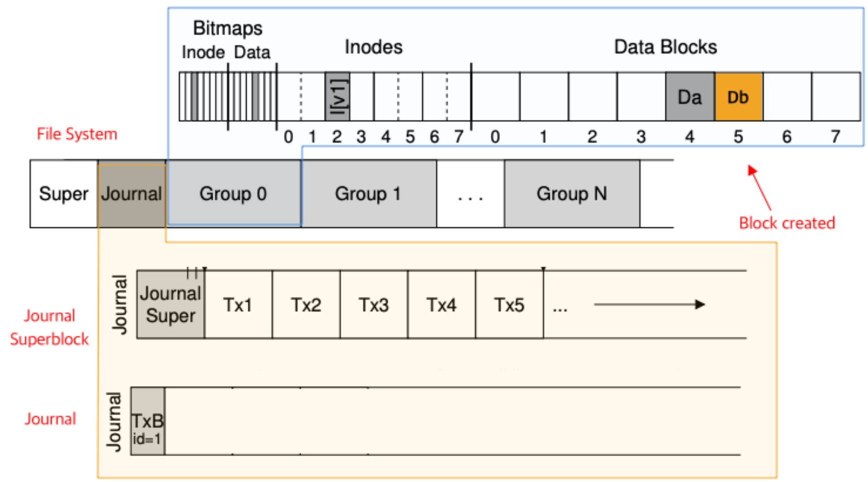
- Goal: Reduce number of writes
- Data block is written to file system first
- Metadata (inode and bitmap information) are written to journal before checkpoint
- Is order dependent
  - e.g. I[v2] and B[v2] make to disk and data block does not
  - If data block is a garbage data, file-system will assume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



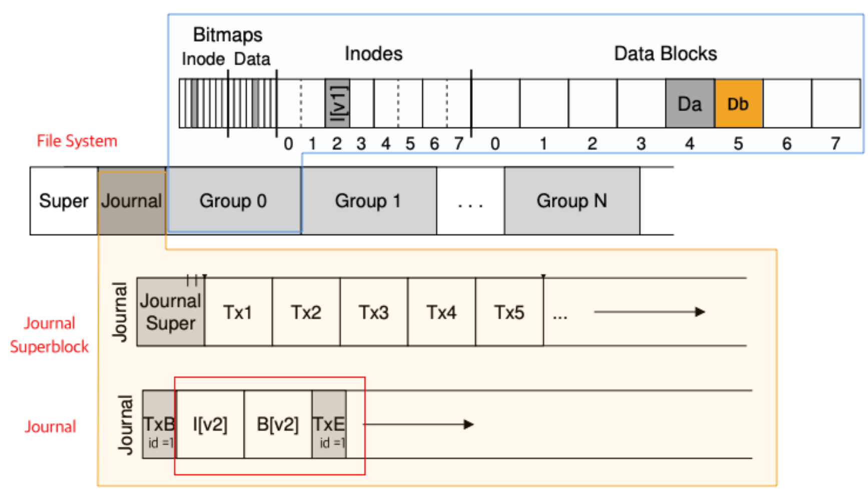
- Steps

	Journal		File System	
TxB	Contents (metadata)	TxE	Metadata	Data
issue	**issue**			**issue**
				complete
complete				
	complete			
		issue		
		complete		
			issue	
			complete	

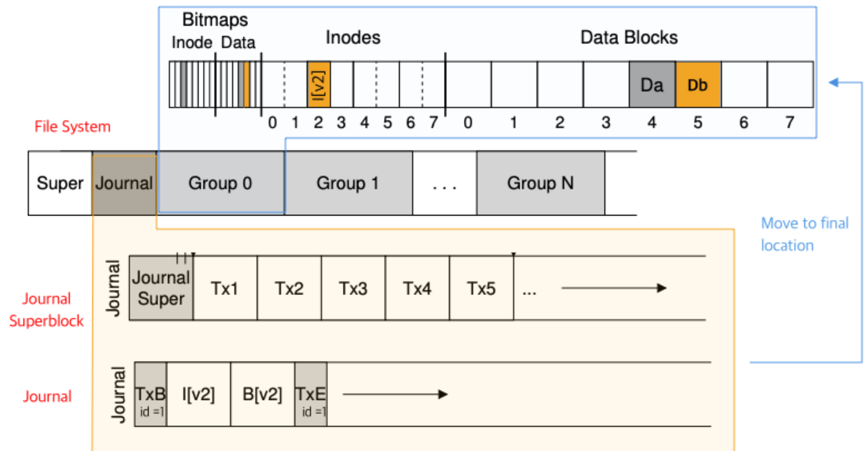
1. **Data Write:** Write data to final location; wait for completion



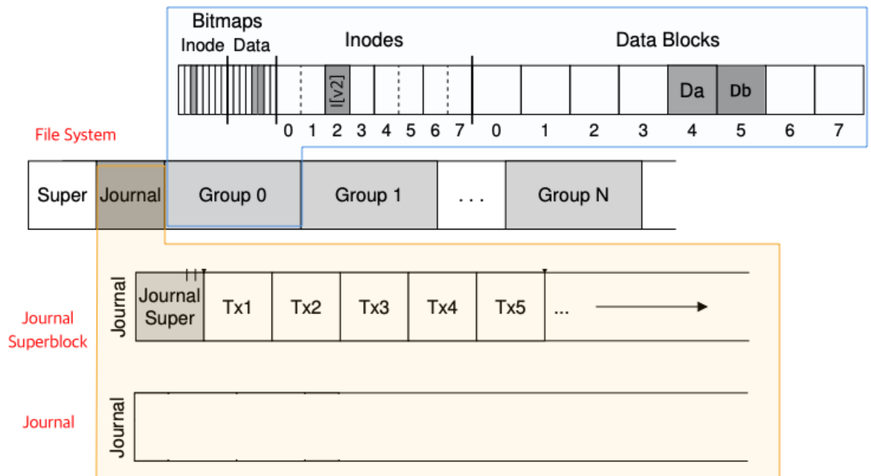
2. **Journal Metadata Write:** Write the begin block and metadata to the log; wait for writes to complete
3. **Journal Commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete



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**