

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	returns 0 if at end
<code>close(fd);</code>	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
<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:**

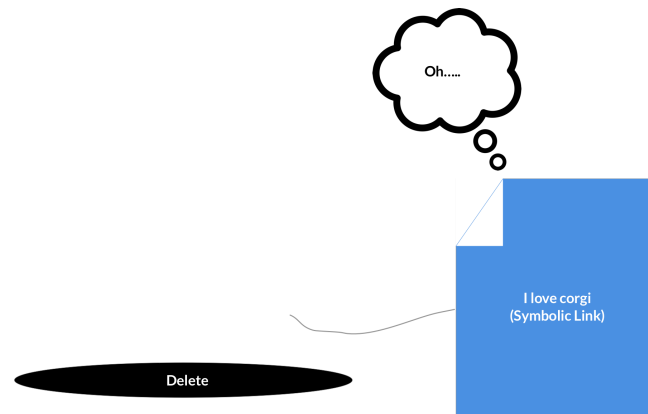
- Is directory entry containing "true" path to the file
- Is a shortcut that reference to a file instead of inode value ^[2]



2



3



- **Hard Link:**

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

- 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

- **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 corresponding 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 * sizeof_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