1. a) 1) 4 - inode blocks. 1 for the file `c`, and 3 for the directdories `/`, `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 pounters), 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`

   b) All of the above

   Notes

   - **Inode**

file contents blocks

file attributes

1
2
3
4
5
6
7
8
9
10
11
12

direct pointers

single indirect
double indirect
triple indirect

pointer block

file contents blocks

file contents blocks

     – 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

c) 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

d) **Rough Work**
- **Creash Scenarios**
  - When only new data block is written to disk
    * This is fine in system's point of view
    * No inode points to it (it doesn't contain any information about file)
    * No bitmap points to it
    * Is as if write never occured
  - When only the updated inode is written to disk
    * There is no bitmap that's pointing to it
    * There is new inode where existing inode is
    * The data block Db hasn't been created
    * Reading data where Db is will return garbage data
    * there is a term for this. Is called **File-System inconsistency**
  - When only inode bitmap is written to disk
    * inode block pointed by bitmap is assumed to be allocated
    * But there is no desired inode where it's pointing
    * This is another example of **File-System-Inconsistency**
    * If left as is, then space cannot be used for future use (**inode leak**)
  - When only data bitmap is written to disk
    * data block pointed by bitmap is assumed to be allocated
    * But there is no desired inode where it's pointing
    * This is another example of **File-System-Inconsistency**
    * If left as is, then space cannot be used for future use (**data leak**)

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
- **File API**
  - open (create/access file)

　　　　* Is a system call
　　　　* 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


```
#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

　　– (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", 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 continues
read(fd, buffer, 100);	100	200	for each call
read(fd, buffer, 100);	100	300	
read(fd, buffer, 100);	0	300	returns 0
close(fd);	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

```c
#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 <u>beginning</u> of file
  * Returns -1 if error

<u>Example</u>

System Calls	Return Code	Current Offset	
`fd = open("file", O_RDONLY);`	3	0	
`lseek(fd, 200, SEEK_SET);`	200	200	move 200 bytes from the start of file
`read(fd, buffer, 50);`	50	250	
`close(fd);`	0	–	read 50 bytes

- `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 <u>file</u> (including symbolic link) from the system
  - **Syntax:** int unlink(const char *pathname)
    * pathname - path to file
  - Returns 0 if successful
  - Returns -1 if error

<u>Example</u>

```
#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

Example

```c
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
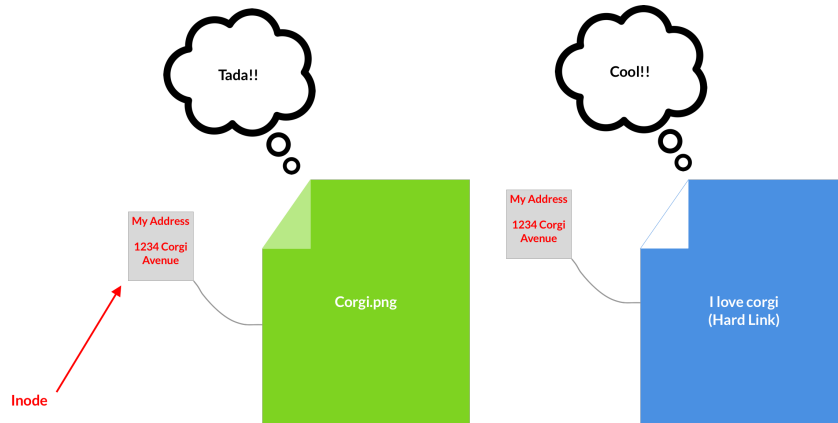 - Is a shortcut that reference to a file instead of inode value [2]
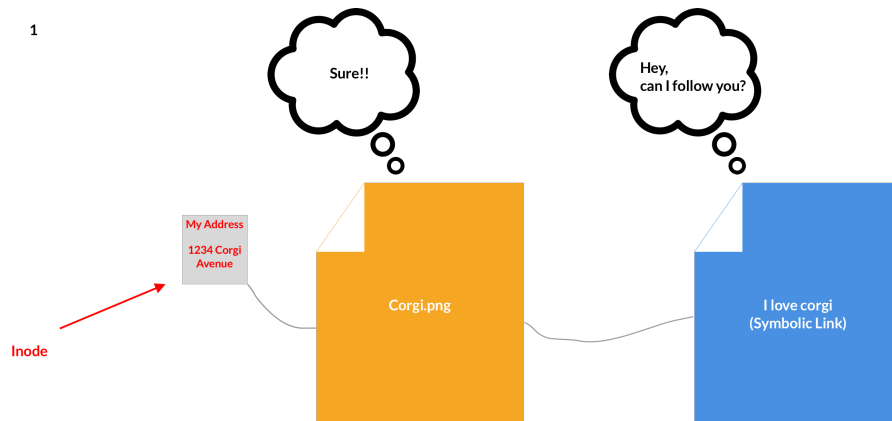
- **Hard Link:**
  - Is a direct reference to a file via its inode [2]
  - Is second directory entry identical to first

References

1) codewiki, stat, link
2) The Open Group Base Specification, unlink, link