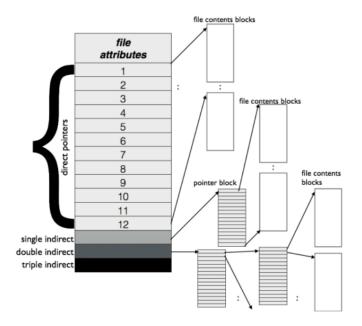
- 1. a) 1) 4 inode blocks. 1 for the file c, and 3 for the directdories /, a, b
 - 2) 3 directory blocks one for root /, one for a, the other for b
 - 3) 1 single indirect block as far as we know. The file definitely has more than 12 blocks (# of data blocks pointed by direct pounters), but less than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



- 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)

- * Is a system call
- * 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 private per process
 - · Can be used to read() and write() files

Example

```
File can also
                                        File can also
                                                                   File can also
                       File can be read
                                        be written by
                                                      be read by
                                                                   be read by
                       by owner
                                                      group
                                                                   others
                                        owner
#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
                              Create file if doesn't exist AND
                              3. Overwrite file if exists
```

- (read)
 - * 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
```

- write
 - * 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", 0_WRONLY | 0_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;
}</pre>
```

- 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)
- \cdot 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 the offset of the pointer (in bytes) from the beginning of file
- Reading and Writing Files
- Reading and Writing Files
- Renaming Files
- Removing Files
- Making Directories
- Reading Directories
- Removing Directories
- Hard Links
- Symbolic Links