CS 111: Operating System Principles

Lecture 15

# Filesystems

2.0.1

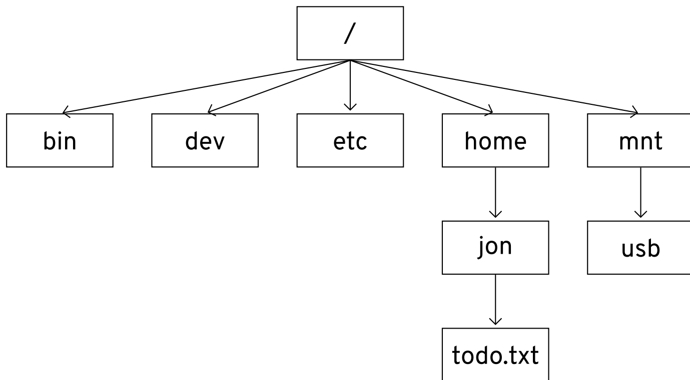
Jon Eyolfson

August 12, 2021

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

# Filesystems

Usual layout of a POSIX Filesystem (here: parts of FHS):




Working Directory: /home/jon

What are the absolute and realtive path to **todo.txt**? To **usb**?

# POSIX Filesystem

todo.txt Relative: ./todo.txt

todo.txt Absolute: /home/jon/todo.txt

usb Relative: ../../mnt/usb

usb Absolute: /mnt/usb

Special symbols:

- . – Current directory
- .. – Parent directory
- $\sim$ – User's home directory ($HOME)

Relative paths are calculated from current working directory ($PWD)

# You Can Access Files Sequentially or Randomly

Sequential access

- Each read advances the position inside the file
- Writes are appended and the position set to the end afterwards

Random access

- Records can be read/written to the file in any order
- A specific position is required for each operation

# POSIX Filesystem

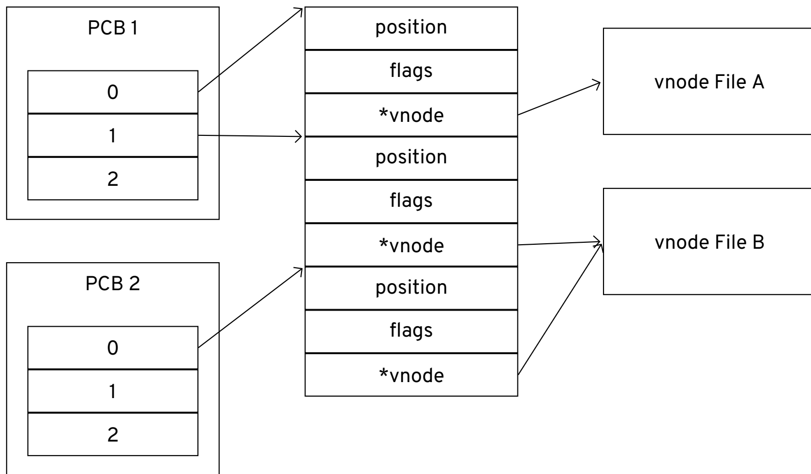
```
int open(const char *pathname, int flags, mode_t mode);

// flags can specify which operations: O_RDWR,O_WRONLY, O_RDWR
// also: O_APPEND moves the position to the end of the file initially

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

// lseek changes the position to the offset
// whence can be one of: SEEK_SET, SEEK_CUR, SEEK_END
//   set makes the offset absolute, cur and end are both relative
```

# Accessing Directory API

```
DIR *opendir(char *path); // open directory
struct dirent *readdir(DIR *dir); // get next item
int closedir(DIR *dir); // close directory
```

```
void print_directory_contents(char *path) {
    DIR *dir = opendir(path);
    struct dirent *item;
    while (item = readdir(dir)) {
        printf("- %s\n", item->d_name);
    }
    closedir(path);
}
```

# File Tables Are Stored in the Process Control Block (PCB)



# Each Process Contains a File Table in its PCB

A File Descriptor is an index in the table

Each item points to a system-wide *global open file table*

The GOF table holds information about the seek position and flags

It also points to a *VNode* (supports read/write/etc)

A vnode (virtual mode) holds information about the file

vnodes can represent regular files, pipes, network sockets, etc.

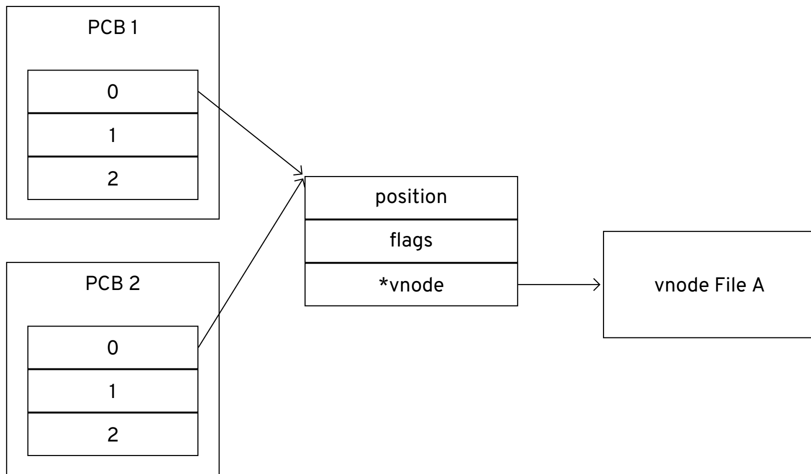
# Remember What Happens In A Fork

PCB is copied on fork

Specifically for us, the local open file table gets inherited

Both PCBs point to the same Global Open File Table entry

# Both Processes Point to the Same GOF Entry



# There Are Some “Gotchas” For This Sharing

Current position in file is shared between both processes

Seek in one process leads to seek in all other processes using the same GOF entry

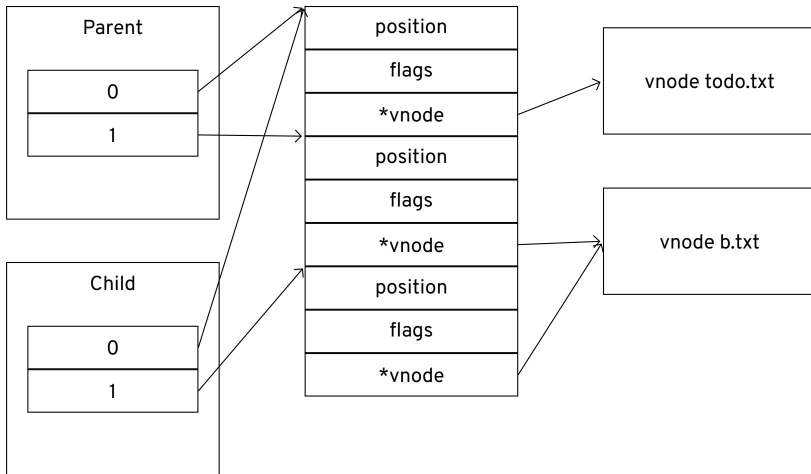
Opening the same file in both processes after forking creates multiple GOF entries

# How many LOF and GOF Entries Exist? What is the Relationship?

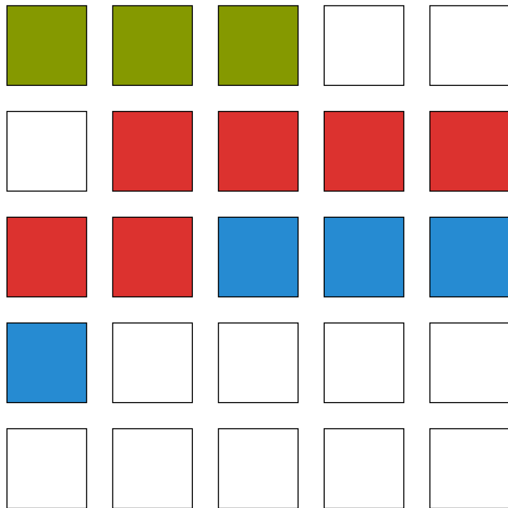
```
open("todo.txt", O_RDONLY);
fork();
open("b.txt", O_RDONLY);
```

Assume there are no previously opened files (not even the standard ones)

# There are 2 LOF Entries Each, and 3 GOF Entries



# How Do We Store Files? Contiguous Allocation?



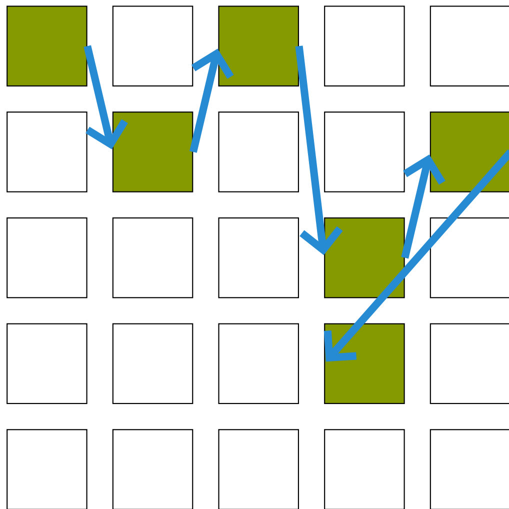
# Contiguous Allocation Is Fast, If There Are No Modifications

Space efficient: Only start block and # of blocks need to be stored

Fast random access: $block = floor(\frac{offset}{blocksize})$

Files can not grow easily
- Internal fragmentation (may not fill a block)
- External fragmentation when files are deleted or truncated

# What About Storing Like a Free List of Pages? Linked Allocation



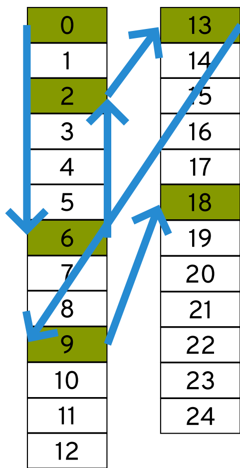
# Linked Allocation Has Slow Random Access

Space efficient: Only start block needs to be stored
- Blocks need to store a pointer to the next block (block is slightly smaller)

Files can grow/shrink
- No external fragmentation
- Internal fragmentation

How can we increase random access speed? We need to walk each block
- Each block may be located far away (it will never be cached)

# File Allocation Table Moves The List to a Separate Table




| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

# File Allocation Table (FAT) is Similar to Linked Allocation

Files can grow/shrink

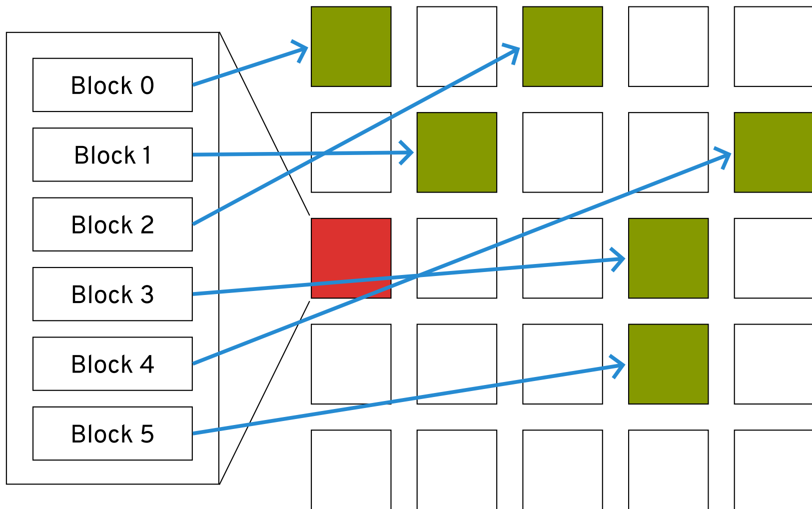
- No external fragmentation
- Internal fragmentation

Fast random access: FAT can be held in memory/cache

- FAT size is linear to disk size: can become very large

How can we further increase random access speed?

# Indexed Allocation Maps Each Block Directly



# For Indexed Allocation, Each File Needs an Index Block

- Files can still grow/shrink
  - No external fragmentation
  - Internal fragmentation
- Fast random access
- File size limited by the maximum size of the index block (fit it in one block)

# Indexed Allocation Problem

Assume this scenario:

- An index block stores pointers to data blocks only (no meta information)
- A disk block is 8 KiB in size
- A pointer to a block is 4 Bytes

What is the maximum size of a file managed by this index block?

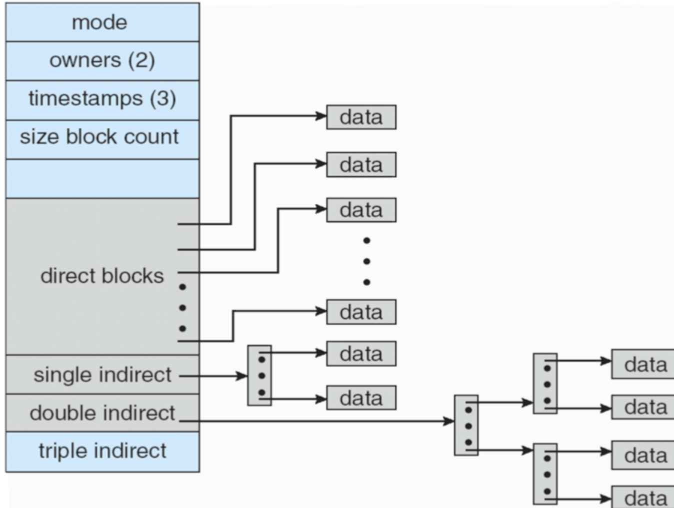
# Indexed Allocation Solution

Assume this scenario:

- An index block stores pointers to data blocks only (no meta information)
- A disk block is 8 KiB in size
- A pointer to a block is 4 Bytes

# of pointers = $\frac{8\text{KiB}}{4\text{B}} \frac{2^{13}\text{B}}{2^{2}\text{B}} = 2^{11}$
# of addressable blocks = # of pointers
Total of bytes = $2^{11} \times 2^{13} = 2^{24} = 16\text{MiB}$

# An inode Describes a File System Object (Files and Directories)



# Linux inodes Aim to Be Efficient for Small Files, and Support Large Ones

UNIX inodes hold metadata and pointers to blocks

Smaller files only use direct pointers

Larger files have additional index nodes with pointers to additional blocks

Very small files can have it’s contents directly inside the inode

# inode Problem

Assume this scenario:

- An index block stores 12 direct pointers, 1 single, double and triple indirect pointer each
- A disk block is 8 KiB in size
- A pointer to a block is 4 Bytes
- Indirect blocks consist of direct pointers only

What is the maximum size of a file managed by this index block?

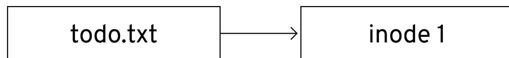
# inode Solution

Assume this scenario:

- An index block stores 12 direct pointers, 1 single, double and triple indirect pointer each
- A disk block is 8 KiB in size
- A pointer to a block is 4 Bytes
- Indirect blocks consist of direct pointers only

# pointers per indirect table = $\frac{2^{13}}{2^{2}} = 2^{11}$
# addressable blocks = $12 + 2^{11} + (2^{11})^2 + (2^{11})^3 \approx (2^{11})^3 = 2^{33}$
Total of bytes = $2^{33} * 2^{13} = 2^{46} = 64 TiB$

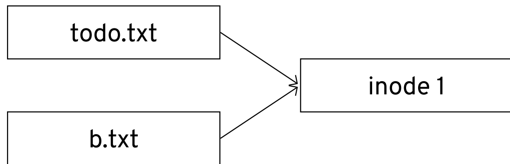
# Hard Links Are Pointers to inodes




A directory entry (aka filename) is called a *hard link*

A hard link points to one inode

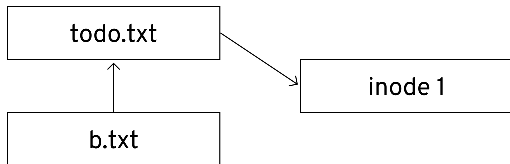
# Multiple Hard Links Can Point to the Same inode




Deleting a file only removes a hard link (the file can be hard liked somewhere else)

POSIX has the `unlink` call rather than a delete call

# Soft Links Are Paths to Another File




When resolving the file, the file system is redirected somewhere else, so:

- Soft link targets do not need to exist
- Soft link targets can be deleted without notice of the soft link
- Unresolvable soft links lead to an exception

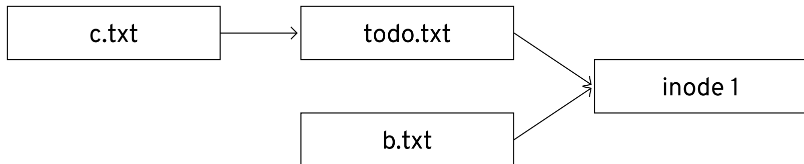
# Filesystem Example Problem

```
touch todo.txt
ln todo.txt b.txt
ln -s todo.txt c.txt
mv todo.txt d.txt
rm b.txt
```

How does the FS look like before and after the mv and rm commands?

# Filesystem Example Solution (1)

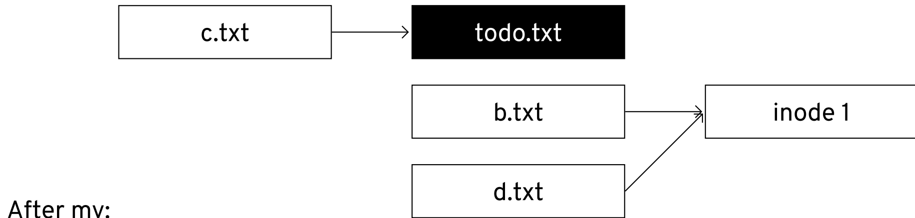
```
touch todo.txt
ln todo.txt b.txt
ln -s todo.txt c.txt
mv todo.txt d.txt
rm b.txt
```




Before mv:

# Filesystem Example Solution (2)

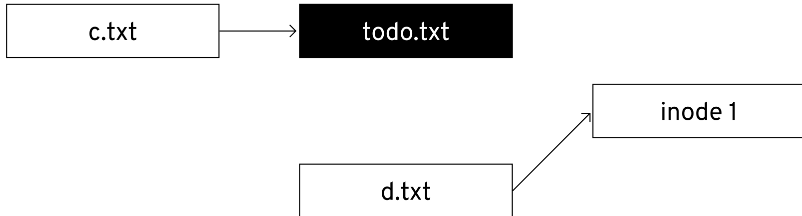
```
touch todo.txt
ln todo.txt b.txt
ln -s todo.txt c.txt
mv todo.txt d.txt
rm b.txt
```



# Filesystem Example Solution (3)

```
touch todo.txt
ln todo.txt b.txt
ln -s todo.txt c.txt
mv todo.txt d.txt
rm b.txt
```

After rm:



# In UNIX, Everything is a File

Directories are files of type “directory”

Additional types are “regular file”, “block device” (HDDs, SSDs), “pipe”, “socket” etc.

Directory inodes do not store pointers to data blocks but rather filenames and pointers to inodes

# What Data is Stored in an inode?

a Filename

b Containing Directory name

c File Size

d File type

e # of soft links to file

f location of soft links

g # of hard links to file

h location of hard links

i access rights

j timestamps

k file contents

l ordered list of data blocks

# What Data is Stored in an inode? Solution

a Filename *No. Names are stored in directories*

b Containing Directory name *No. File can be in multiple dirs*

c File Size *Yes*

d File type *Yes*

e # of soft links to file *No (they are unknown)*

f location of soft links *No (they are unknown)*

g # of hard links to file *Yes (to know when to erase the file, check* `stat`*)*

h location of hard links *No (they are unknown to the inode)*

i access rights *Yes*

j timestamps *Yes*

k file contents *Sometimes*

l ordered list of data blocks *Yes, by definition*

# Filesystem Caches Speed Up Writing to Disks

Writing data to the disk is slow, we can use a cache to speed it up

File blocks are cached in main memory in the *filesystem cache*

- Referenced blocks are likely to be referenced again (temporal locality)
- Logically near blocks are likely to be referenced (spatial locality)

A kernel thread (or daemon) writes changes periodically to disk

`flush` and `sync` system calls trigger a permanent write

# Journaling Filesystem

Deleting a file on a Unix file system involves three steps:

1. Removing its directory entry.
2. Releasing the inode to the pool of free inodes.
3. Returning all disk blocks to the pool of free disk blocks.

Crashes could occur between any steps, leading to a storage leak

The journal contains operations in progress, so if a crash occurs we can recover

# Filesystems Enable Persistence

They describe how files are stored on disks:

- API-wise you can open files, and change the position to read/write at
- Each process has a local open file and there's a global open file table
- There's multiple allocation strategies: contiguous, linked, FAT, indexed
- Linux uses a hybrid inode approach
- Everything is a file on UNIX, names in a directory can be hard or soft links