Please sit close to the front of the class to help me save my voice =)

Please sit close to someone if possible for in-class discussions.

File Systems

2024 Semester 2 COMPSCI 340: Operating Systems

Lecture 1

Talia Xu

1.0.0

File system APIs

Open

- Open-file table: tracks open files
- File descriptor: pointer to last read/write location
- File-open count: counter of number of times a file is open
- Disk location of the file: cache of data access information
- Access rights: per-process access mode information

Close

- close() decreases the open count
- When the open count reaches zero, the file is no longer in use - allow removal of data from open-file table when last processes closes it
- create() and delete() are system calls that work with closed rather than open files

POSIX Filesystem

```c
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

int close(int fd);

// close() closes a file descriptor, so that it no longer
// refers to any file and may be reused.

// returns 0 on success.
```

File system APIs

Create

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

```
$ strace touch empty
execve("/usr/bin/touch", ["touch", "empty"], 0x7ffec0b50ca8 /* 62 vars */) = 0
...
openat(AT_FDCWD, "empty", O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK, 0666) = 3
...
utimensat(0, NULL, NULL, 0) = 0
```

File system APIs

Delete

```
int unlink(const char *pathname);
```

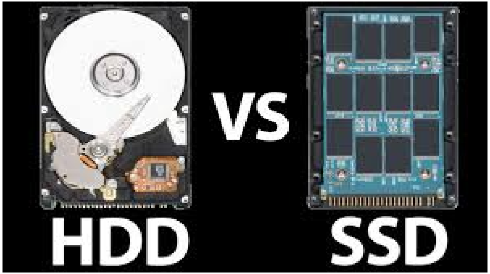
```
$ strace rm empty_5mb.txt
openatexecve("/usr/bin/rm", ["rm", "empty_5mb.txt"], 0x7ffec6aa3788 /* 62 vars */) = 0
...

newfstatat(AT_FDCWD, "empty_5mb.txt", {st_mode=S_IFREG|0664, st_size=5242880, ...}, AT_SYMLINK_NOFOLLOW) = 0
faccessat2(AT_FDCWD, "empty_5mb.txt", W_OK, AT_EACCESS) = 0
unlinkat(AT_FDCWD, "empty_5mb.txt", 0) = 0
```

Is the deleted data overwritten? Is data recovery possible?

Aside: What is a solid state drive (SSD) ?

Use transistors (like RAM) to store data rather than magnetic disks.



SSDs are more modern

Pros

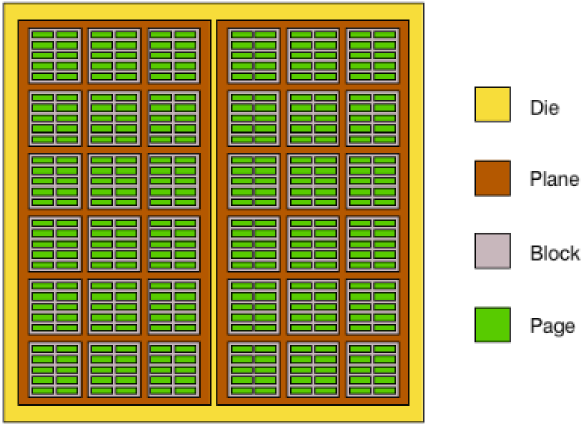
- No moving parts or physical limitations
- Higher throughput, and good random access
- More energy efficient
- Better space density

Cons

- More expensive
- Lower endurance (number of writes)
- More complicated to write drivers for

A SSD contains pages

Pages are typically 4 KB.



NAND Flash Programming Uses Pages and Blocks

You can only read complete pages and write to freshly erased pages

Erasing is done per block (a block has 128 or 256 pages)

- An entire block needs to be erased before writing

Writing is slow (may need to create a new block)

The OS Can Help Speed Up SSDs

SSDs need to garbage collect blocks

- Move any pages that are still alive to a new block (may be overhead)

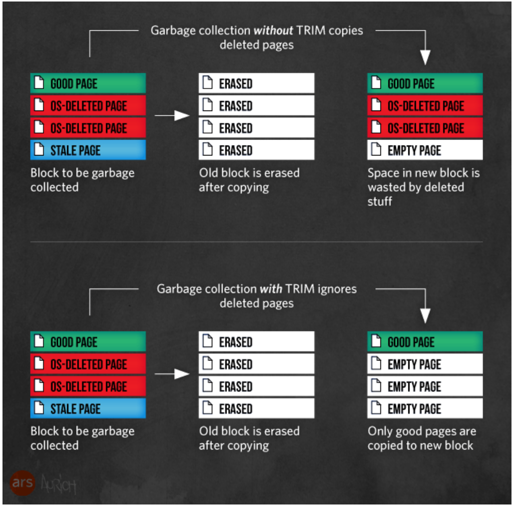
The disk controller doesn't know what blocks are still alive

- SSD may think the disk is full, when a file could be deleted (not erased)

The OS can use the **TRIM** command to inform the SSD a block is unused

- The SSD can freely erase the block without moving overhead

TRIM



Garbage collection *without* TRIM copies deleted pages
GOOD PAGE
OS-DELETED PAGE
OS-DELETED PAGE
STALE PAGE
Block to be garbage collected
ERASED
ERASED
ERASED
ERASED
Old block is erased after copying
GOOD PAGE
OS-DELETED PAGE
OS-DELETED PAGE
EMPTY PAGE
Space in new block is wasted by deleted stuff
Garbage collection *with* TRIM ignores deleted pages
GOOD PAGE
OS-DELETED PAGE
OS-DELETED PAGE
STALE PAGE
Block to be garbage collected
ERASED
ERASED
ERASED
ERASED
Old block is erased after copying
GOOD PAGE
EMPTY PAGE
EMPTY PAGE
EMPTY PAGE
Only good pages are copied to new block
ars

Back to the API: delete()

Most modern computers use SDD.

Most modern OS has TRIM enabled by default.

Direct recovery is near impossible.

- Versioning file system

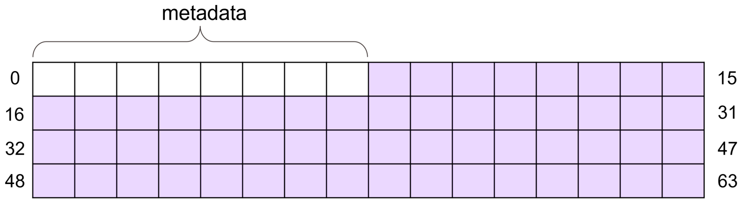
Storing files on disk

We have learned about creating and deleting files from the user side.

How is this implemented on the disk?

We will start by looking at a very simple file system (VSFS).

A Very Simple File System (VSFS)



A disk is divided into a series of blocks. A common size is 4 KB (or 8).

What do we need to store in these blocks to build a file system?

In the beginning, every block is “empty” (not allocated).

Example metadata

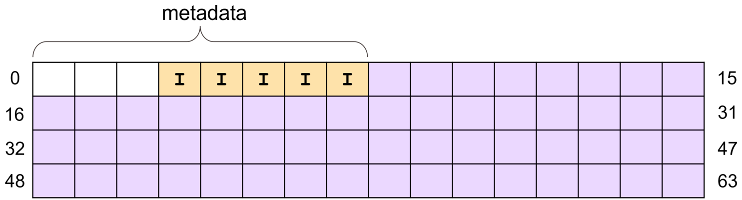
Core Attributes

- Filename: The name used to identify the file.
- File Type: Indicates the type of file (e.g., text, image, video, executable, etc.).
- File Size: The size of the file in bytes.
- Permissions: Controls who can read, write, and execute the file (user, group, others).
- Ownership: Identifies the user and group that own the file.

Timestamps

- Creation Time: When the file was created.
- Last Modified Time: When the file's content was last changed.
- Last Access Time: When the file was last opened or read.

A Very Simple File System (VSFS)



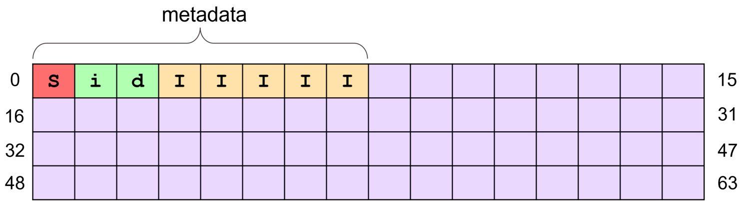
Metadata is stored in i-nodes (more on this later).

i-nodes are typically 128 bytes or *256 bytes*.
Each block is 4 KB.

How many i-nodes can we have in our VSFS?

- What does this mean?

A Very Simple File System (VSFS)



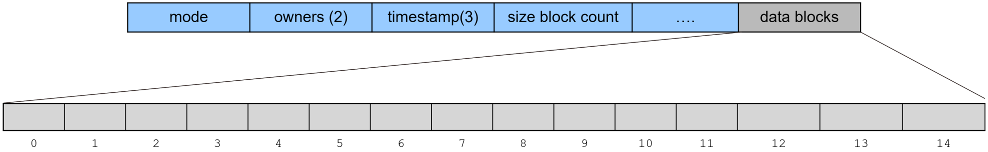
Which i-node and data blocks are free?

- A bit map: each bit indicates whether the corresponding block is free (0) or in use (1)

Superblock

- Number of inodes and data blocks
- Where does the inode table (array) begin?
- What file system is this?

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

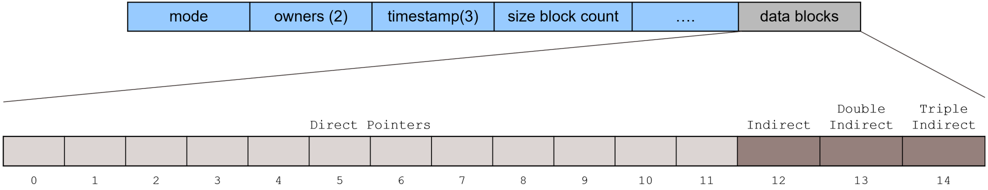


There are 15 (disk) pointers (not memory!)

Each disk block is 4 KB.

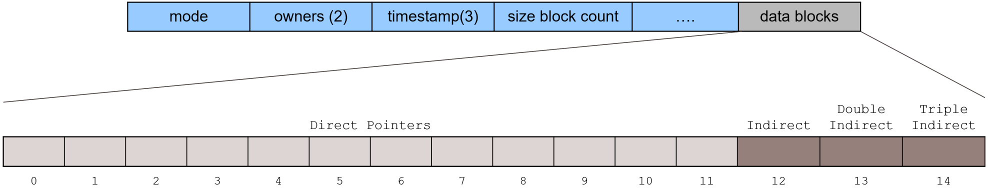
If each pointer points to a disk block. What is the maximum file size?

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



Indirect pointer: Instead of pointing to a block that contains user data, it points to a block that contains more pointers.

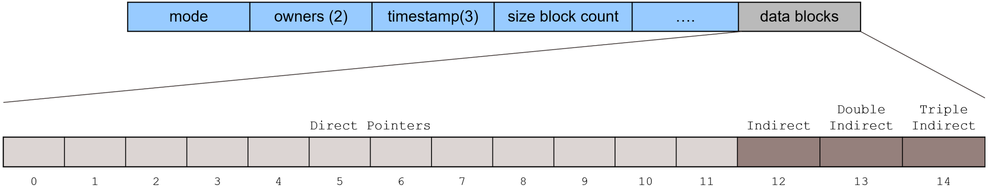
An inode Describes a File System Object (Files and Directories)



Assume each data block is 4 KB.
Assume each pointer is 4 bytes.

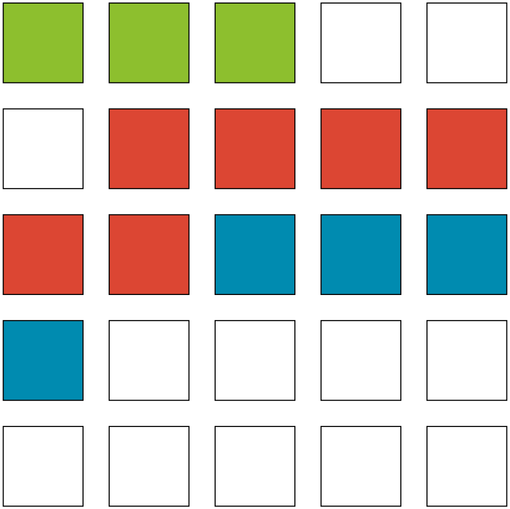
What is the maximum file size of 12 direct pointers + 1 indirect pointer?

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



What is the maximum supported file size?

How Do We Store Files? Contiguous Allocation?



Contiguous Allocation Is Fast, If There Are No Modifications

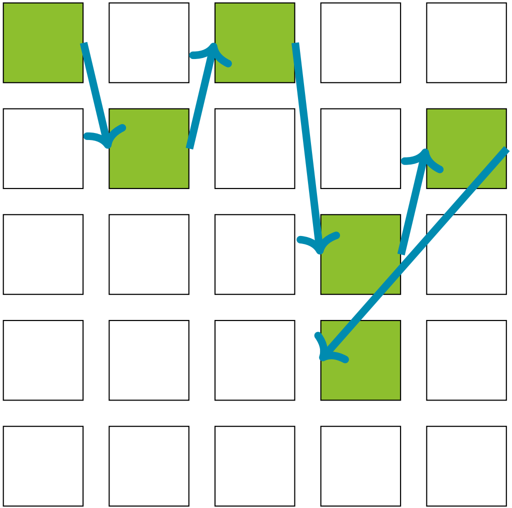
Space efficient: Only start block and # of blocks need to be stored

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)

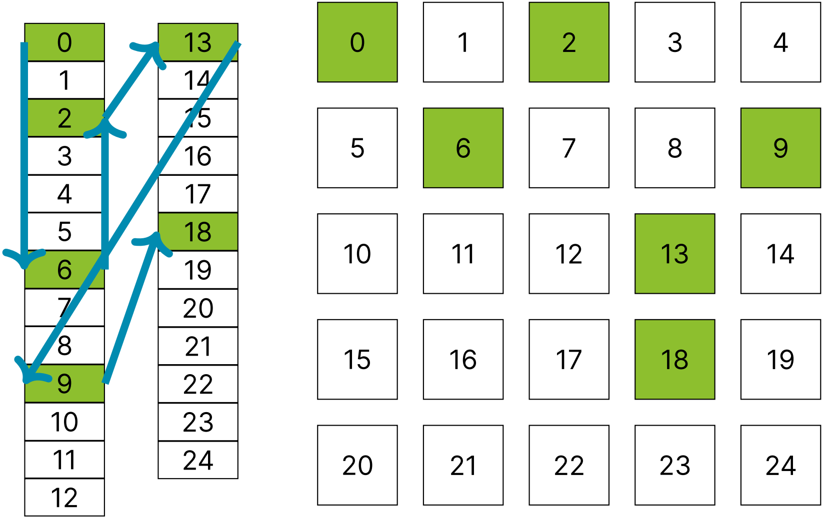
Files can grow/shrink

- No external fragmentation
-

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



File Allocation Table (FAT) is Similar to Linked Allocation

Files can grow/shrink

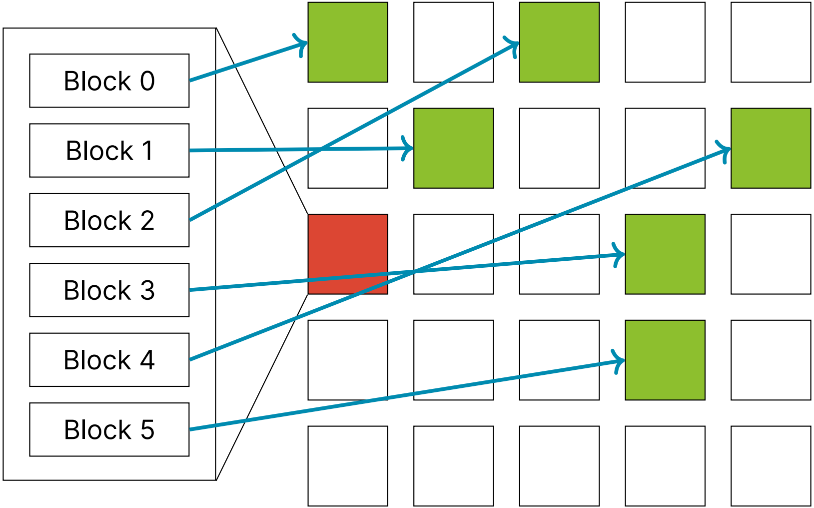
- No external fragmentation
- Internal fragmentation

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)

Reading

Textbook

- 14.4 Allocation Methods