Project 6

File System

Sean Ogden

Slide heritage: Robert Escriva

Cornell CS 4411, November 22, 2013

What do you have to do?

- Implement a virtual file system using a virtual block device.
- Implement a UNIX-like interface to the filesystem.
- You need to support:
  - Create files of **variable size** (while efficiently using disk space)
  - Reclaim storage from deleted files.
  - A hierarchy of nested directories.
  - Concurrent access to the SAME file(s) by multiple threads.

Our proposed plan of attack

- Play with the block device, and see how it behaves.
- Learn the UNIX filesystem API:
  - Parameters.
  - Semantics.
- Decide on details of filesystem internals.
  - Representation of directories, inodes, the superblock, etc..
- Implement.
- Perform extensive testing.
  - Concurrent operations should be your focus when testing.

What is this virtual block device?

- Stores blocks in a regular NT file.
  - Creating a disk $\Rightarrow$ create the NT file.
  - Spin-up a disk $\Rightarrow$ open the NT file.
- We support just one disk which will contain all of your MINIFILESYSTEM.
- The block device is just a raw stream of bytes with no organization.
  - Need to create any and all structures on top of the block device.

How does this block device work?

- Block-level operations
  - Read/write take a block-sized buffer, and block number.
  - Blocks are hard coded using `DISK_BLOCK_SIZE`.
- Operations are asynchronous (just like a real device).
  - You schedule requests by a control call to the device.
    - A **limited number of requests** may be processed at any one time.
    - Requests will be arbitrarily delayed and **re-ordered**.
  - Asynchronous events complete by means of an interrupt.
    - Once again, you'll need to **write an interrupt handler**.

Creating/attaching a block device

```
// Create a new disk
int disk_create(disk_t* disk,
                char* name,
                int size,
                int flags);
// Access an existing disk
int disk_startup(disk_t* disk,
                 char* name);
```

- Flags: `DISK_READWRITE` or `DISK_READONLY`
- Both prepare the disk for use.

Sending requests to the disk

```
int disk_send_request(disk_t* disk,
        int blocknum, char* buffer,
        disk_request_type_t type);
```

- Request types:
  - `DISK_RESET` Cancel any pending requests.
  - `DISK_SHUTDOWN` Flush buffers and shutdown the device.
  - `DISK_READ` Read a single block.
  - `DISK_WRITE` Write a single block.
- Responses are returned asynchronously.
- Return values: 0 = success, -1 = error, -2 = too many requests.
- Wrappers: `disk_read_block` / `disk_write_block`.

The interrupt handler

Install it with:

```
void install_disk_handler(
        interrupt_handler_t disk_handler);
```

Argument you will receive:

```
typedef struct {
    disk_t* disk;
    disk_request_t request;
    disk_reply_t reply;
} disk_interrupt_arg_t;
```

Interrupt notification types

`DISK_REPLY_OK` operation succeeded.

`DISK_REPLY_FAILED` disk failed on this request for no apparent reason.

`DISK_REPLY_ERROR` disk nonexistent or block outside disk requested.

`DISK_REPLY_CRASHED` it happens occasionally.

The API you will write I

- Creation (`creat`) / deletion (`unlink`)
- Open / close
  - Modes are similar to `fopen` in UNIX.
  - Sequential reading, writing (with truncation), appending.
  - Any reasonable combination of the above.
- Read or write a chunk of data (for an open file)
  - Position in the file is unspecified (by the API), and operations are sequential.
  - Chunk size may be any size, and is not related to block size.
  - Blocking operations, which return when completed or failed.
  - Short reads: when not enough data is present completely fulfill the request.

The semantics of files

- Only sequential access (no `fseek`);
  - Read from the begin to end.
  - Writes at the beginning, but cause truncation.
  - Appending starts at the end of a file.
  - Writing/appending causes files to adjust to accommodate the data.
- Assume binary data (that is, don't assume null-termination or newlines).
- Concurrent access.
  - A notion of a "cursor" that indicates read / write position.
    - A **separate** cursor is maintained for each open file.
  - Restrictions apply. See below for semantics.
- Stat returns the file size for files.

The semantics of directories

- Creation and deletion of directories affects the filesystem.
- Change and get current directory.
  - `Current directory` is a local, per-process parameter.
  - Does not affect the filesystem.
- List contents of the current directory.
- Stat returns -2 for directories.

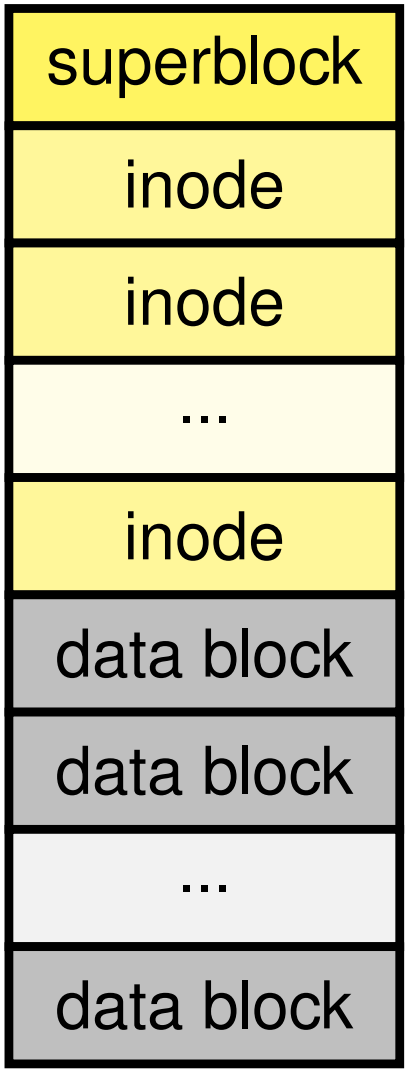
The API I

```
minifile_t minifile_creat(char *filename);
minifile_t minifile_open(char *filename,
                         char *mode);
int minifile_read(minifile_t file,
                  char *data,
                  int maxlen);
int minifile_write(minifile_t file,
                   char *data,
                   int len);
int minifile_close(minifile_t file);
int minifile_unlink(char *filename);
int minifile_mkdir(char *dirname);
int minifile_rmdir(char *dirname);
```

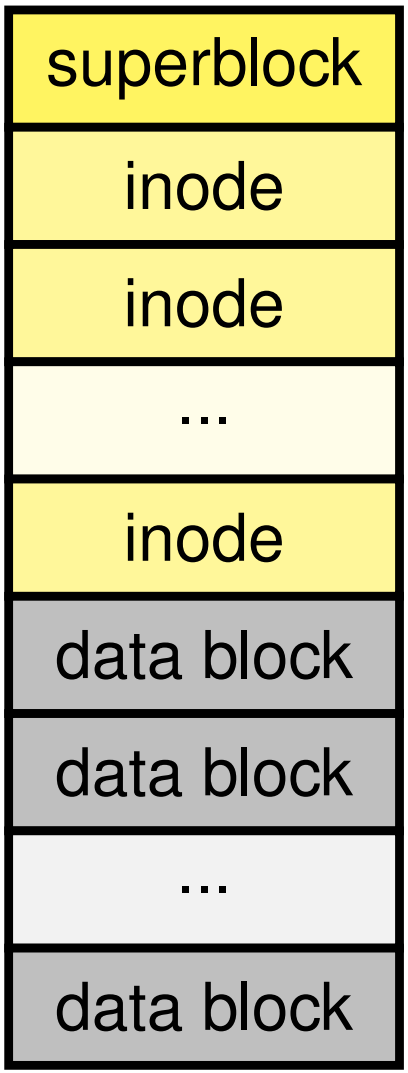
The API II

```
int minifile_stat(char *path);
int minifile_cd(char *path);
char **minifile_ls(char *path);
char* minifile_pwd(void);
```

A picture is worth 1,000 words



The superblock

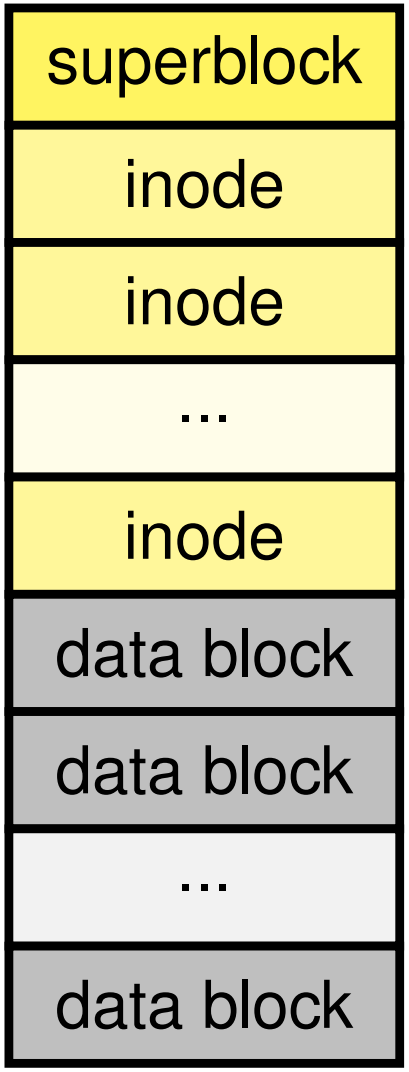


- Points to the root inode (the `'/'` directory).
- Points to the first free inode[a].
- Points to the first free data block[b]
- Holds statistics about the filesystem:
  - Number of free inodes and blocks.
  - Overall filesystem size.
- Contains a magic number (the first four bytes)
  - Helps to detect a legitimate file system.

---

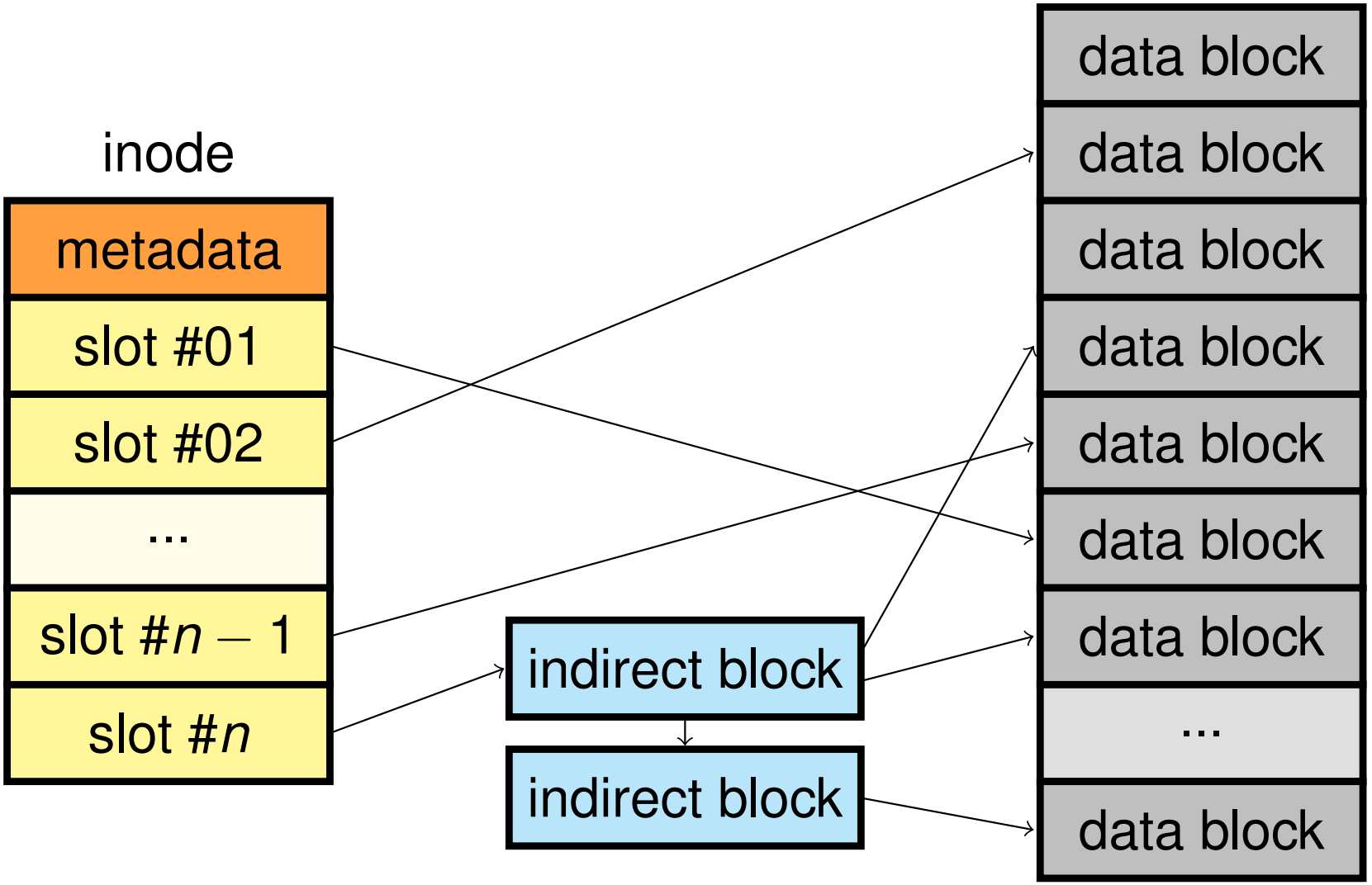
[a] if the free inodes form a linked list
[b] if the free data blocks form a linked list

The inodes

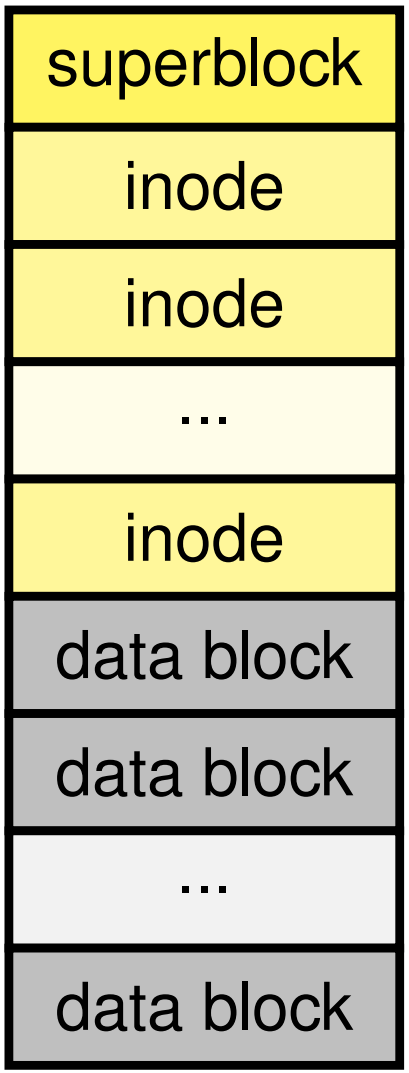


- Cumulatively occupy $\sim$ 10% of disk space.
- Hold information about the file and directory:
  - Metadata, including type, size, etc..
  - Data blocks occupied by the file.
    - A few are directly addressable.
    - A single block is an indirect block.

Block chaining

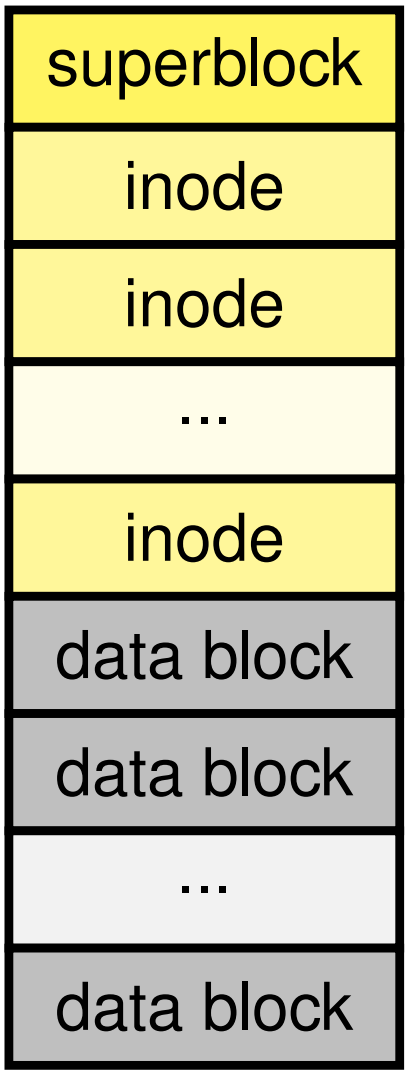


Files



- Files are binary.
- Stored directly in blocks.

Directories



- A special, fixed format (your decision).
  - Can be either ASCII or binary.
- Per-file entries:
  - Name (at least 256 characters).
  - inode number (for the “main” inode).
- Don’t bother with fancy structures; instead, do a linear search.

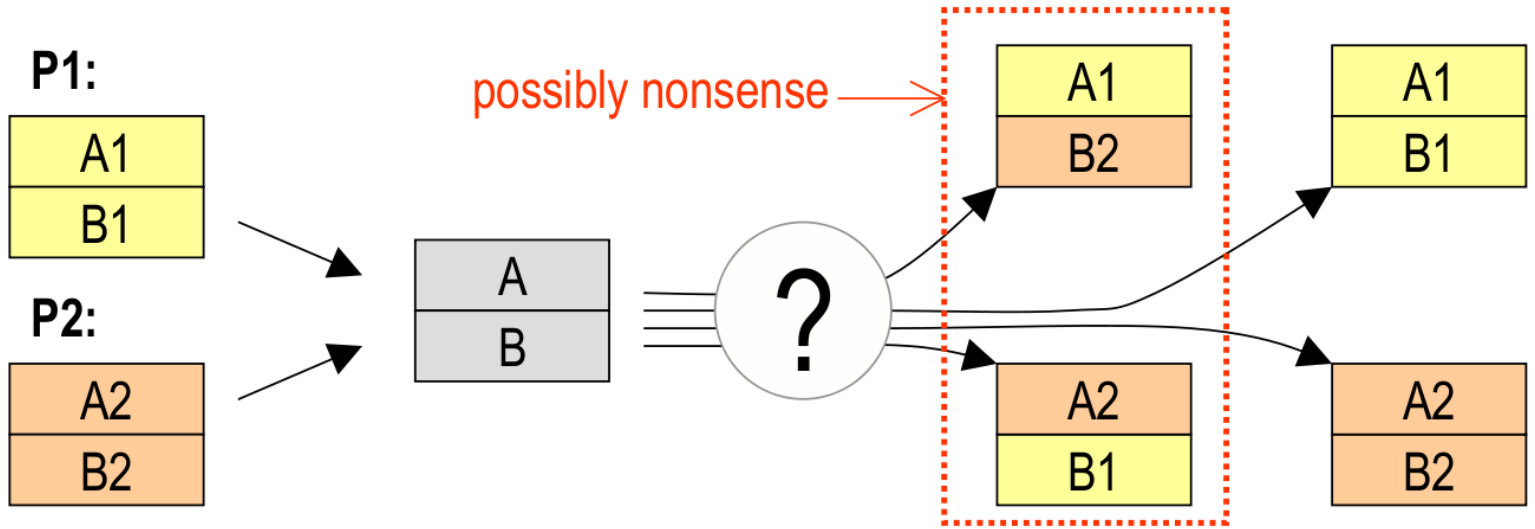
UNIX vs. Windows

We present both for the sake of disclosing the ways in which real operating systems work.

You should follow the UNIX way.

UNIX read/write semantics I

- Allow multiple writers to the same file.
- Don't give guarantees about the integrity of files.
  - The result of concurrent writes may be a mix of both writes ...
  - .. which may not represent anything sensible.



- Consistent with the *end-to-end principle*.

UNIX read/write semantics II

- Simple ... but there are some traps that lead to integrity issues.
  - Cannot just naively overwrite inodes as this leads to orphaned data blocks.
  - You need consistent, synchronized metadata updates.

Alternative approaches*: multi-read / single-write

- Concurrency happens at the “data blocks” level.
- Multiple individuals may hold the same file.
- Opening for writing always succeeds.
- At most one writer works simultaneously.
- Multi-block atomicity: let’s throw the end-to-end argument out the window.

*Alternative, as in we are not grading to these specifications

Alternative approaches: Windows semantics†

- Either multiple readers **OR** a (single at most) writer.
- Exclusion is enforced at the time files are being opened.
- Hold-and-wait springs to mind.

† A.K.A. You can do better than this

Deletion

- UNIX semantics[‡].
  - File is immediately unopenable (e.g. removed from directory structures).
  - Blocks are not placed onto the free list immediately.
  - Applications using the file are unaffected.
  - Actually delete when the last application closes the file.
  - Recycling blocks requires reference counting.
  - Changes made after deletion are lost.
- Windows semantics
  - If the file is being read or written, the deletion fails.

[‡] We will be looking at this when grading

Keep interfaces the same

- Do not change the APIs (this causes build errors).
- Just reporting an error is sufficient (no need to make error codes).

Correctness

- Disk controllers will reorder requests
- Need to handle crashes smoothly:
  - Ctrl+C: disk should be in a consistent state.
  - Disk crashes: don't issue any more requests to it.
- We will test failure conditions:

```
extern double crash_rate;
extern double failure_rate;
extern double reordering_rate;
```

Efficiency

- Start with simple data structures (e.g. a single inode per block).
- Focus on correctness.
  - Breath-taking performance won't help if your system doesn't work as specified.[§]
  - Premature optimization is the root of all evil (Knuth)
  - Do fancy things later

[§] Also known as, "I don't care that your Ferarri is fast if it only does left-hand turns."

Source Code

`disk.h/cc` The virtual block device.

`shell.c` A sample shell to work with.

`minifile.h` Prototypes for the functions we ask you to implement.

`minifile.c` Your implementation.

Testing

- Use the supplied shell program.
- Write your own tests!
  - Not just variations on the same theme.
  - Think outside the box.
  - Concurrent updates can be tricky.
  - Write a **`fsck`** program to verify the consistency of your filesystem.
  - Check the correctness of the written data.
    - We'll check against UNIX semantics.
  - What happens as the crash/failure/reorder rates tend toward 1.0?

General guidelines

- Split the development into several pieces:
  - Create the disk structure.
  - Verifying the disk structure (this should be an easy-to-call routine).
    - Not crashing is not good enough.
  - Create/navigate directories.
    - Create inodes, and directory structure.
    - Track each thread's current directory.
  - Create and delete files.
    - Single process first, then add synchronization.
  - Reading/writing, truncating/enlarging
    - Single process, maintaining the cursor.
    - Add synchronization, testing with multiple readers/writers.

Concluding thoughts

- Have some fun with this project.
- This project is a change of pace (no more networking!)
- Come see the TAs in office hours.