

Project 6

File Systems

Zhiyuan Teo

Slide heritage: Previous TAs → Robert Escriva

Cornell CS 4411, November 18, 2011

Administrative Information

- Project 3 has been regraded, please check your scores.
- Project 4 has been graded. Use CMS for regrade requests.
 - We are swamped right now, give us some time to sort out all regrade requests.
- Project 5 is due Sunday, 20 November at 11:59PM.
There will be no extension for this project.
- Project 6 will be released this evening, due December 5 at 11:59PM.

- 1 Project Scope
- 2 Implementation Details
- 3 File System Structure
- 4 Concurrent Access
- 5 Considerations

1 Project Scope

2 Implementation Details

3 File System Structure

4 Concurrent Access

5 Considerations

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 operations that:
 - 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.
 - Simplification: only sequential access, no random accesses.

A proposed plan of attack

- Understand the block device and 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.
 - Start with basic operations that are single-threaded.
 - Then stress your system by testing it under concurrent workload.

- 1 Project Scope
- 2 Implementation Details**
- 3 File System Structure
- 4 Concurrent Access
- 5 Considerations

What is this virtual block device?

- Stores blocks in a regular Windows file.
 - Initialize a disk \Rightarrow creates a Windows file or opens an existing one.
- We support just one disk which will contain all of your MINIFILESYSTEM.
- The device provides disk access at the block granularity, but 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 takes a block-sized buffer and block number.
 - Block size is defined as a constant: `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**.

Initializing the block device

```
// Initialize the disk  
int disk_initialize(disk_t* disk);
```

- Inspects some global variables and either creates a new disk or uses an existing one.
- **This has to be called before installing the disk handler.**
 - Or else you will get mysterious crashes and/or "Error 6".

Global variables for `disk_initialize()`

- Set the following global flags before calling `disk_initialize()`:
 - `const char* disk_name`: provides the name of the Windows file used to store your virtual disk.
 - `int use_existing_disk`: set to 1 if using an existing disk, 0 to create a new one.
 - `int disk_flags`: set to `DISK_READWRITE` or `DISK_READONLY`.
 - `int disk_size`: number of blocks allocated for the disk.
- These flags are described in `disk.h`.

Some important instructions

- If `use_existing_flags` is set to 0, this creates the disk given by `disk_name`).
 - The parameters `disk_name`, `disk_size` and `disk_flags` will be used to create the disk.
- If `use_existing_disk` is set to 1, this starts up an existing disk given by `disk_name`.
 - After starting up, `disk_size` and `disk_flags` will be automatically updated.

Some important instructions

- Why use these flags?
- Long story short: **you will lose many, many points if you do not do this!**
 - A substantial portion of our autograder will test the persistence of your filesystem across simulated reboots.
 - If you create a new disk everytime you start minithreads, tests will fail to show the persistence.
- Set these global flags from your user application.
 - Set `use_existing_disk` to 0 in `mkfs`
 - Other applications should set this to 1.

Sending requests to the disk

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

■ Request types:

`DISK_READ` Read a single block.

`DISK_WRITE` Write a single block.

`DISK_RESET` Cancel any pending requests.

`DISK_SHUTDOWN` Flush buffers and shutdown the device.

■ Return values: 0 = success, -1 = error, -2 = too many requests.

Sending requests to the disk

- `disk_send_request()` returns the status of queueing the operation.
- Wrappers for commonly-used functions:
 - `disk_read_block()` Fetches a block from the disk.
 - `disk_write_block()` Writes a block to disk.
- Actual disk response is returned asynchronously via interrupts.

Disk 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;
```

Free up this argument when you are done with it.

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 is outside disk requested.

`DISK_REPLY_CRASHED` it happens occasionally.

The API you will write: file related

```
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);
```

The API you will write: directory related

```
int minifile_mkdir(char *dirname);  
int minifile_rmdir(char *dirname);  
int minifile_cd(char *path);  
char* minifile_pwd(void);
```

The API you will write: miscellaneous

```
int minifile_stat(char *path);  
char **minifile_ls(char *path);
```

Opening a file

- Creation (`creat`) / deletion (`unlink`)
- Opening a file:
 - Modes are similar to `fopen` in UNIX.
 - `r`, `w`, `a`, `r+`, `w+`, `a+`
 - Check man pages for `fopen` to find out more about these modes.
 - Sequential reading, writing (with truncation), appending.
 - Any reasonable combination of the above.

File reading and writing

- Reading/writing a chunk of data (for an open file):
 - Position in the file cannot be specified through the API; operations are sequential.
 - Chunk size may be any size, and is not related to block size.
 - Operations block and return only when completed or failed.
 - Short reads: when not enough data is present to completely fulfill the request (should only happen at end-of-file).

File semantics

- A "cursor" is maintained for **each** open file handle (`minifile_t`).
 - Cursor indicates the next read / write position.
- Only sequential access (no `fseek`):
 - Reads go from the beginning to the end.
 - Writes start at the beginning, but cause truncation.
 - Appends start at the end of a file.
 - Writing/appending causes files to adjust to accommodate the data.
 - Position your cursor accordingly.

Other details about files

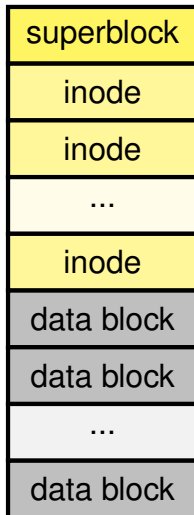
- Assume binary data (that is, don't assume null-termination or newlines).
- Concurrent access.
 - Multiple minithreads can concurrently read/write/delete. More on this later.
- `stat` returns the file size for files.

Directory semantics

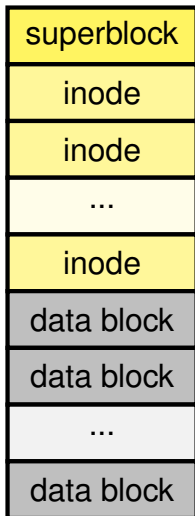
- Creation and deletion of directories affects the file system, but:
- Changing and getting current directory does not.
 - Current directory is a local, **per-minithread** parameter.
- `ls` lists contents of the current directory.
 - Return an array of `char*`
 - Each entry is a null-terminated string.
 - Last item in this array should be a `NULL`.
- `stat` returns -2 for directories.

- 1 Project Scope
- 2 Implementation Details
- 3 File System Structure**
- 4 Concurrent Access
- 5 Considerations

The grand view



The superblock

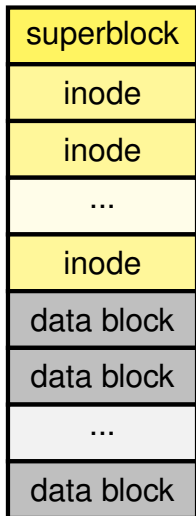


- Contains a magic number (the first four bytes)
 - Helps to detect a legitimate file system.
- Points to the root inode (the `'/'` directory).
- Points to the first free inode^a.
- Points to the first free data block^b.
- Contains statistics about the filesystem:
 - Number of free inodes and blocks.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

^aif the free inodes form a linked list

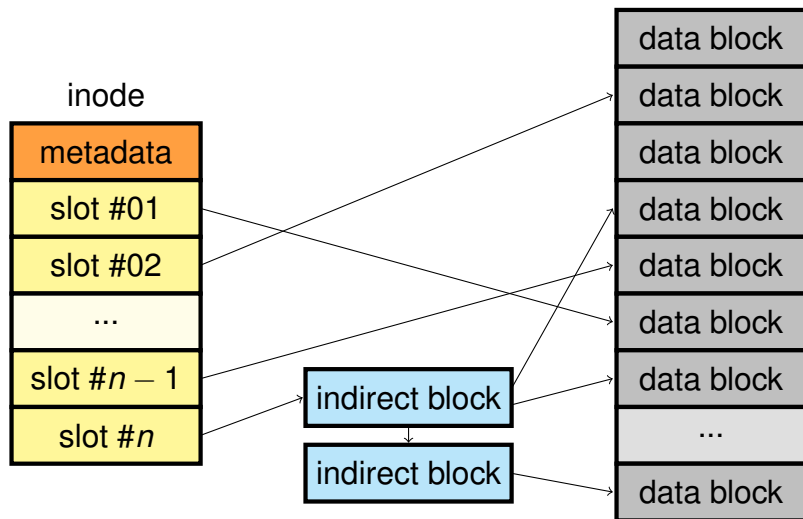
^bif the free data blocks form a linked list

inodes

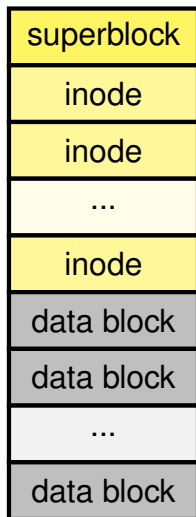


- Cumulatively occupy $\sim 10\%$ of disk space.
- Each inode holds information about the file or directory:
 - Metadata, including type, size, etc.
 - Contains some slots that point directly to data blocks.
 - Last slot is an indirect block.
 - An indirect block is a data block that contains even more slots.
- Does not contain file or directory names.

Block chaining

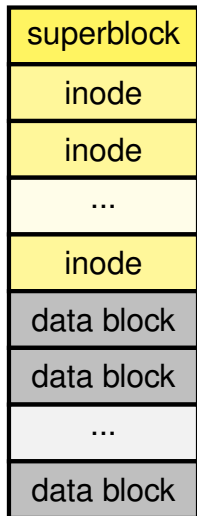


Files



- Files are binary.
- Stored in data blocks.

Directories



- Each directory inode has at least 1 corresponding data block.
- These data blocks contain file/dir names and their inode mappings.
- Format of directory data blocks:
 - Name (at least 256 characters).
 - Corresponding inode number.
- The first two entries should be "." and ".."
- Don't bother with fancy structures; instead, do a linear search.

- 1 Project Scope
- 2 Implementation Details
- 3 File System Structure
- 4 Concurrent Access**
- 5 Considerations

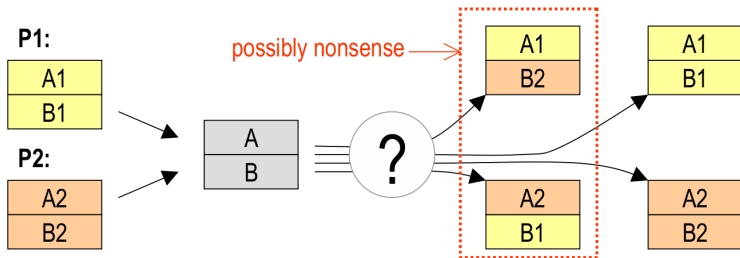
UNIX vs. Windows

We present both for the sake of disclosing the ways in which real operating systems work. You should* follow the UNIX way.

*must

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

- Conceptually 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, but we are not grading to these specifications

Alternative approaches: Windows semantics[‡]

- Either multiple readers **or** a single 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 (ie. 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 expect UNIX semantics when grading

- 1 Project Scope
- 2 Implementation Details
- 3 File System Structure
- 4 Concurrent Access
- 5 Considerations**

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

- Use 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 Ferrari is fast if it only does left-hand turns."

Source Code

`disk.h/c` 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 correctness of the written data.
 - We'll check against UNIX semantics.
 - What happens as the crash/failure/reorder rates tend toward 1.0?

Suggestions

- Split the development into several pieces:
 - Begin with `mkfs`, create the disk structure.
 - Write a `fsck` file system verifier before you begin, so you are clear about the disk structure in complex cases.
 - Create/navigate directories.
 - Create inodes and data blocks for directory structure.
 - Track each thread's current directory.
 - Create and delete files.
 - Single minithread first, then add synchronization.
 - Reading/writing, truncating/enlarging.
 - Maintain the cursor for each file handle.
 - Add synchronization, testing with multiple readers/writers.

Concluding thoughts

- This is the last project!
- Begin early so you have time to study for the finals.
- Come see the TAs in office hours.