CS415 Project #6: File System

Krzysztof Ostrowski
krzys@cs.cornell.edu

What do you have to do?

- Implement a virtual file system
  - On top of a raw virtual block device provided by us
    - Storing all blocks of the virtual disk device in a single file
    - Single filesystem on single device, without mount points etc.
  - With a UNIX-like interface
  - With support for:
    - Creation of files of **variable size** (using disk space efficiently)
    - Reclaiming unused storage from deleted files
    - A hierarchy of nested directories
    - Concurrent access to the SAME files by multiple threads

Sequence of actions

Development plan:

- Get familiar with the block device you get
- Get familar with the API you need to cover
  - Together with the parameters and semantics
- Decide on details of disk organization
  - How are directories kept, inodes, superblock etc.
- Decide on semantics with concurrent access
- Implement
- Perform extensive testing
  - In particular, concurrent operations on files

What do you get?

- Our virtual device
  - Block are kept in a regular NT file
    - Our disk can also be "created", "spinned-up" etc.
      - ...which corresponds to a file being created or opened.
    - We support just one disk, in a file MINIFILESYSTEM
      - Attach it and spin it up as a part of system startup
  - A raw stream of bytes: no organization
    - Need your own structures: i-nodes, free blocks etc.
    - Need to create any such structures on disk yourself
      - Write a system tool "mkfs.exe" or auto-create on startup

What do you get?

- Our virtual device
  - Supports block-level operations
    - Specify block number + provide a buffer to read/write
      - Block size is fixed to 4K, hard-coded into the system
  - Works asynchronously (just like a real device)
    - You schedule requests by a control call to the device
      - A **limited number of requests** may be processed at a time!
      - Requests can be arbitrarily delayed and **re-ordered**, need to take reordering into account e.g. when appending data
    - Notification is received as an interrupt
      - We let you register a **special type of interrupt handler**

Our virtual block device

- Creating a new virtual disk

```
int disk_create(disk_t* disk, char* name,
   int size, int flags);
```

  - creates a disk with a given "**`name`**" (in a given NT file)
  - flags: **`DISK_READWRITE`** or **`DISK_READONLY`**
  - actually, size and flags are stored in the file...
    - ...so the disk "remembers" this information

- Accessing an existing disk

```
int disk_startup(disk_t* disk, char* name);
```

  - returns a handle to the disk with a given "**`name`**"

Our virtual block device

- Sending requests to the device

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

  - request types:

`DISK_RESET`	-- cancel any pending requests etc.
`DISK_SHUTDOWN`	-- flush buffers / shutdown the device
`DISK_READ`	-- read a single block
`DISK_WRITE`	-- write a single block

  - requests are handled asynchronously
  - returns 0 if success, -1 on error, -2 if too many requests
  - wrappers: **`disk_read_block`** / **`disk_write_block`**

Our virtual block device

- Interrupt handler
  - As usual, you need to install your own:

```
install_disk_handler(
                    interrupt_handler_t handler);
```

  - Arguments passed to the handler:

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

← **See the next page!**

Our virtual block device

- Notification received in the interrupt:

`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

What do you provide?

- Files:
  - Creation / deletion ("unlink")
  - Open (an existing file in a specific mode) / close
    - Modes are more or less as in "fopen" in UNIX
    - Sequential reading, writing (w. truncation), appending
    - Any reasonable combinations of all the above
  - Read or write a chunk of data (for an open file)
    - Position in file unspecified, operations are sequential
    - Of any size, not necessarily a multiple of block size
    - Blocking operations, return when completed or failed
    - But: may read less data than requested (if not there)

What do you provide?

- Files:
  - Only sequential access (no "fseek")
    - Reading starts from the beginning, proceeds to end
    - Writing likewise + also causes the file to be truncated
    - Appending starts at the end of the existing file
    - Writing / appending causes the file to be "enlarged"
  - Binary
    - Don’t assume 0-terminated strings, newlines etc.
  - Concurrent access
    - A notion of "cursor" that indicates read / write position
      - A **separate cursor** is maintained for each thread
    - Restrictions apply, choose semantics (see below)

What do you provide?

- Directories:
  - Creation and deletion – affects the filesystem
  - Change and get current directory
    - **Current directory** is a local, per-process parameter
      - No global variables here!
    - Does not have any effect on the filesystem
  - List contents of the current directory
- General:
  - Check status of an object (file / directory)
    - Whether directory or a regular file
    - ...and if regular file, what is its current size

The API you need to cover

```
minifile_t minifile_creat(char *filename);
minifile_t minifile_open(
  char *filename, char *mode);
```

- argument "`mode`" is treated in the same way as in "fopen"

```
int minifile_read(
  minifile_t file, char *data, int maxlen);
int minifile_write(
  minifile_t file, char *data, int len);
```

- "`read`" / "`write`" return the actual num. of bytes read/written

```
int minifile_close(minifile_t);
int minifile_unlink(char *filename);
```

- "`unlink`" deletes the specified file

The API you need to cover

```
int minifile_mkdir(char *dirname);
int minifile_rmdir(char *dirname);
```

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

- check the type (regular file / directory) and size of given file

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

```
char *minifile_pwd();
```

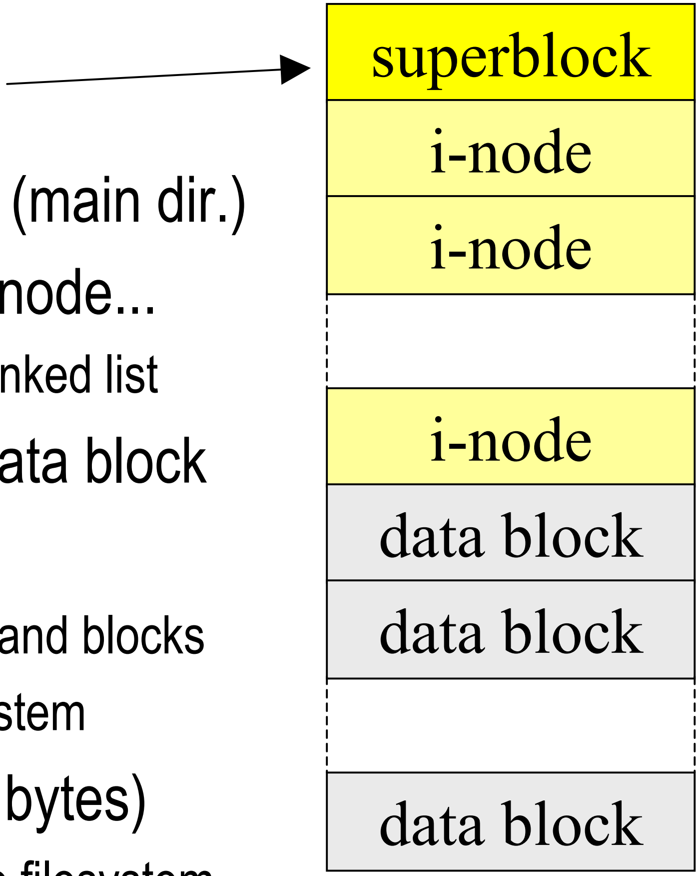
- return the current dir. (the path to it) for the calling thread

- Paths as usually in UNIX-like systems

/dir$_1$/dir$_2$/ … /dir$_n$/filename

Disk organization

- General structure
  - Superblock (global info)
    - Pointer to the root inode (main dir.)
    - Pointer to the first free i-node...
      - ...if free i-nodes form a linked list
    - Pointer to the first free data block
    - Statistics
      - Numbers of free inodes and blocks
      - Overall size of the filesystem
    - Magic number (first four bytes)
      - Helps detect a legitimate filesystem



superblock

i-node

i-node

i-node

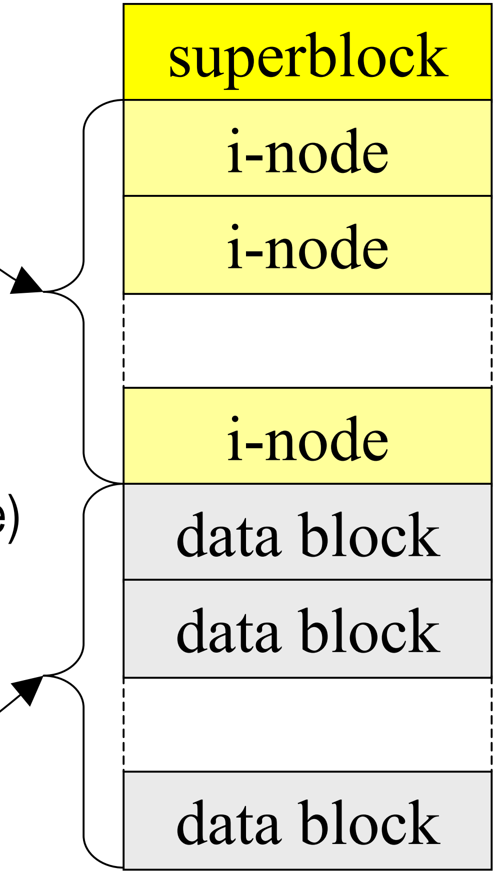
data block

data block

data block

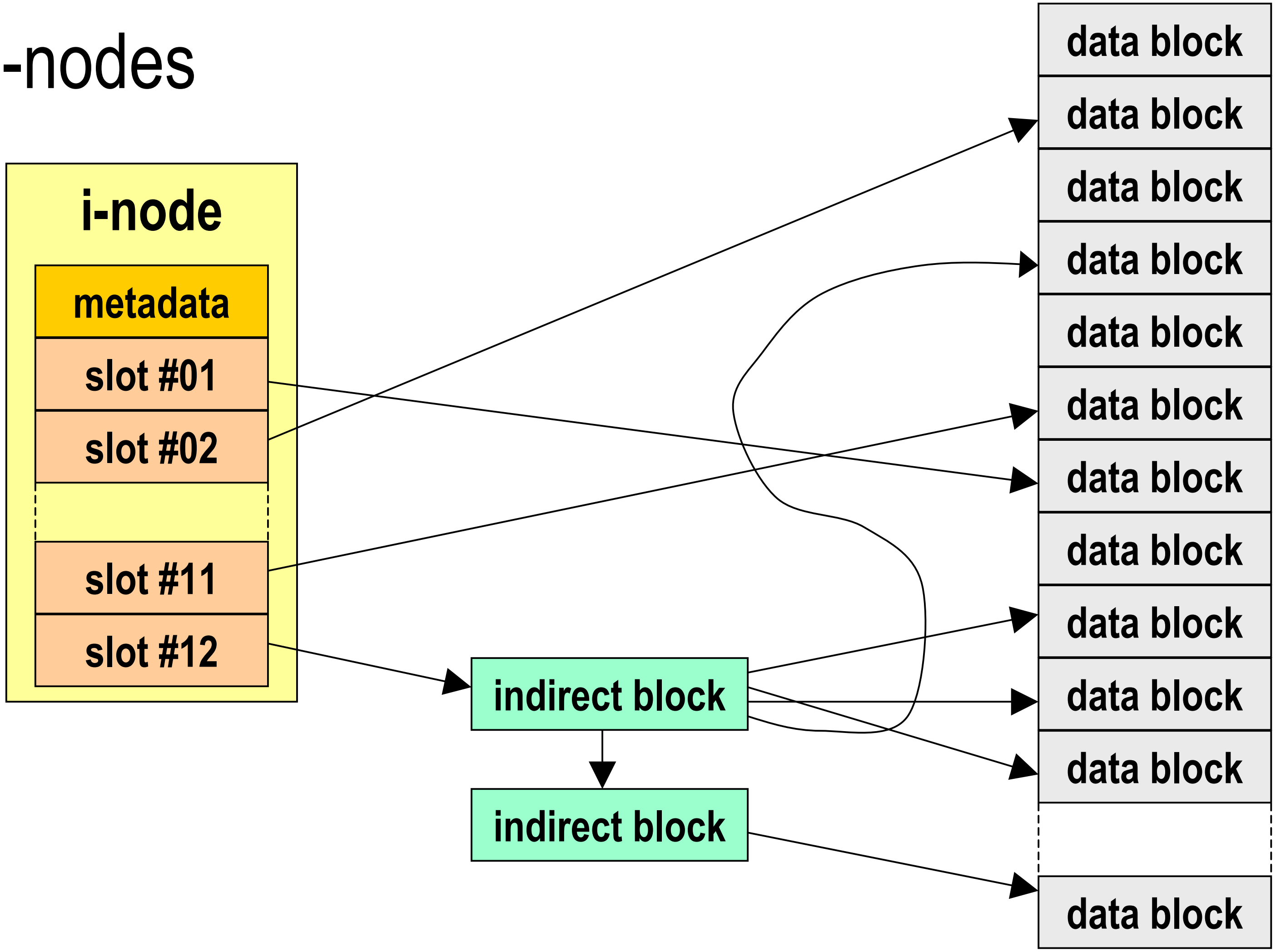
Disk organization

- General structure
  - i-nodes
    - Occupy ~ 10% of disk space
    - All information about file / dir.
      - Metadata, including type (file/dir.), size, next i-node on the list etc.
      - Name: the only exception (not here)
      - Data blocks occupied by the file
        - A few (11) addressed directly
        - A single indirect block
  - Data blocks



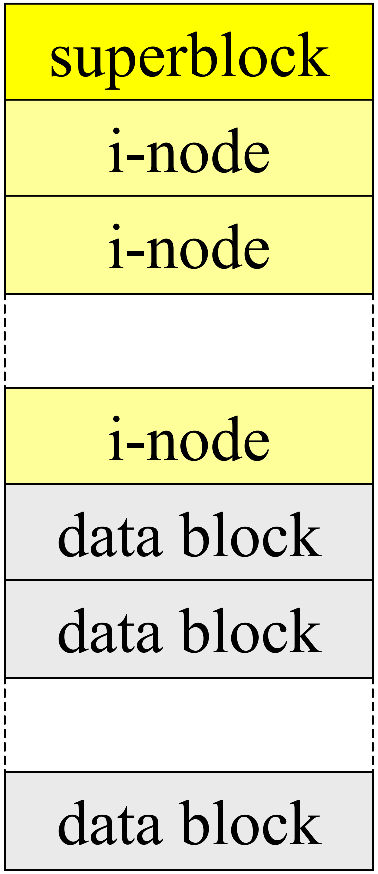
Disk organization

- i-nodes



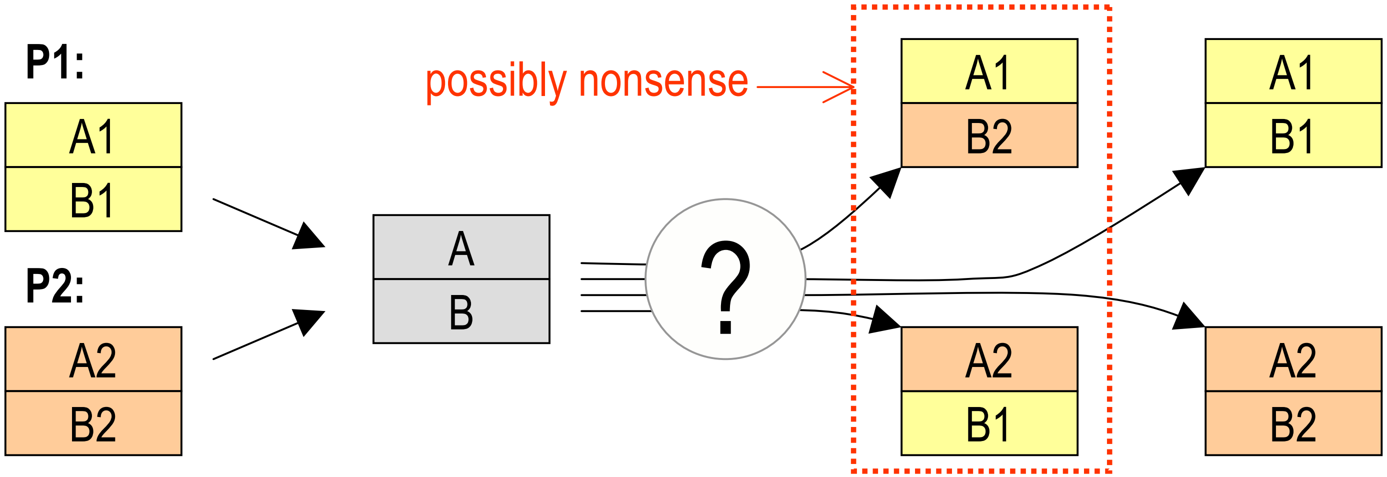
Disk organization

- Data blocks
  - Files: binary, directly in blocks
  - Directories:
    - A special, fixed format (you choose)
      - Can be either ASCII or binary
    - Entries per file:
      - name (allow for at least 256 characters)
      - i-node number (for the "main" i-node)
    - A special type (DIRECTORY)
      - But: keep types in i-nodes, not here
    - Don’t bother about fancy structures
      - Assume just a linear search for a file



Concurrent access

- Read / write : three approaches
  - Approach #1: Unix Semantics (much preferred)
    - Allow multiple writers to the same file
    - Don’t give any guarantees about the integrity of files
      - The result of concurrent writes may be a mix of both writes...
      - ...which in general may not represent anything sensible



Concurrent access

- Approach #1 (continued)
  - Argument in favor if this method: *end-to-end principle*
  - Simple... but: need to preserve integrity of the FS!
    - Cannot just use a naive write that just overwrites i-nodes... ...as this could lead to generation of orphaned data blocks
    - So you need consistent, synchronized metadata updates!
- Approach #2: Multiple Readers / Single Writer
  - Concurrency semantics at the "data blocks" level
    - Multiple readers and writers can open **the SAME file**...
    - ...and hold usable handles, open for write **never blocks**
    - Actual read/write synchronized: at most one writer
    - Multi-block atomicity: avoids problems of the first approach

Concurrent access

- Approach #3: Windows Semantics
  - Either multiple readers **<u>OR</u>** a (<u>single</u> at most) writer
    - Enforced at the time files are being **opened**
    - Quite restrictive: applications may keep unused resources!
  - Arguably easiest, but not recommended

Concurrent access

- Access and deletion
  - Approach #1: Windows Semantics
    - Deletion fails when file is currently being read / written
  - Approach #2: Unix Semantics (much preferred)
    - File is immediately made unusable
      - Removed immediately from directory structures...
      - ...but its blocks are not placed on the free list yet
    - Applications using the file operate unaffected
    - As soon as the last application closes, actually delete
      - Need to keep reference count of open handles
      - Last applicaton to close the file actually recycles its blocks
      - All changes made after deletion are lost

Implementation issues

- Interfaces:
  - Don’t change APIs in any way (need for testing)
  - Don’t need to report detailed error codes
- Correctness:
  - Since disk controllers may reorder requests...
    - Can’t issue concurrent requests for blocks that are to be written sequentially (need to wait)
  - Need to handle crashes smoothly:
    - Ctrl+C: system should be left in **consistent** state
    - Disk crashes: don’t issue any more requests to it

Implementation issues

- Efficiency:
  - Don’t go overly complex with data structures
    - A single i-node per block highly recommended, for access speed as well as overall simplicity
  - Correctness is more important
    - Breath-taking performance won’t help if your system doesn’t work as specified...
      - ...so be conservative with optimizations: basic things first...
      - ...and leave any fancy enhancements for the very end of it!

Source Files

- Provided by us
  - The virtual block device
    `disk.h` / `disk.c`
  - A simple shell for testing purposes
    `shell.c`
- For you to implement
  - The filesystem layer
    `minifile.h` / `minifile.c`

Testing

- You can test with the supplied shell program
  - Create dirs, navigate, list, read/write files etc.
- But: you should write your own tests as well!
  - Try reading and writing small or large files
  - Try concurrent access by multiple threads
    - This is probably the hardest test of all, don’t omit it
  - Do **verify consistency** of your filesystem!
  - Check correctness of the written data...
    - ...according to the semantics you chose to support.
  - Test if you handle disk/system crashes properly

General guidelines

- Make sure scheduler / synchronization work!
- Split all development process into little steps:
  - Creating / verifying overall structure of the disk
    - Needed anyway to do any testing
    - Don't know if your stuff really works if you don't verify
      - ...the absence of visible errors is **not** a proof of corectness!
  - Directories
    - Creating an i-node + creating a directory structure
    - Adding a per-process path to "current directory", then navigating
  - Creating / deleting files
    - Single process first (implement + test), then add synchronization
  - Reading / writing, truncating / enlarging
    - Start from a single process, maintain cursor etc.
    - Add synchronization, test with multiple readers and writers