ReFS Starter Code

CS333 :: Storage Systems

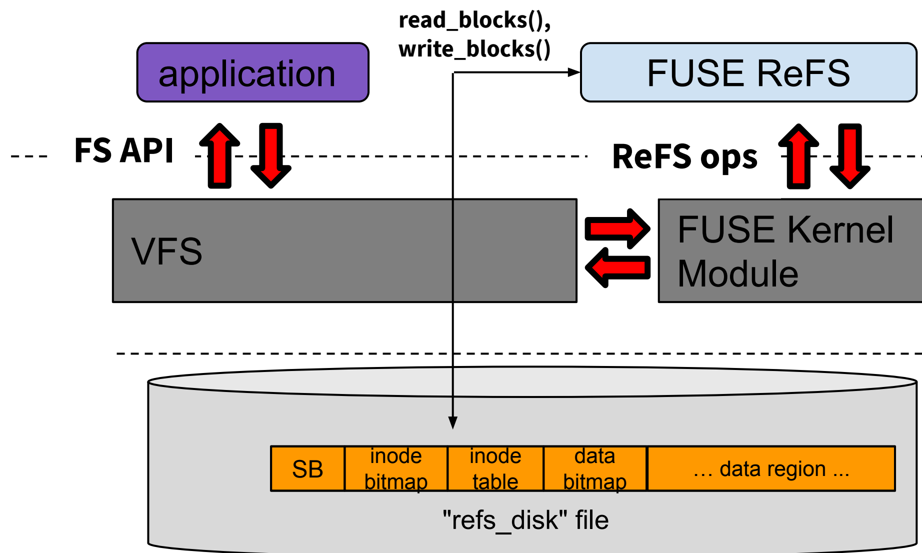
Williams College

ReFS Design

- Largely based on the reference file system in OSTEP ch. 40
- The differences are minor, and were made with the goal of simplifying our FS implementation:
  - Bitmaps are located immediately next to structures they track
    - See slide "Disk Format"
  - Directory data blocks use fixed-size entries, letting us treat directory data blocks as "arrays of entries"
    - See slide "Directory Structure"

FUSE infrastructure & organization

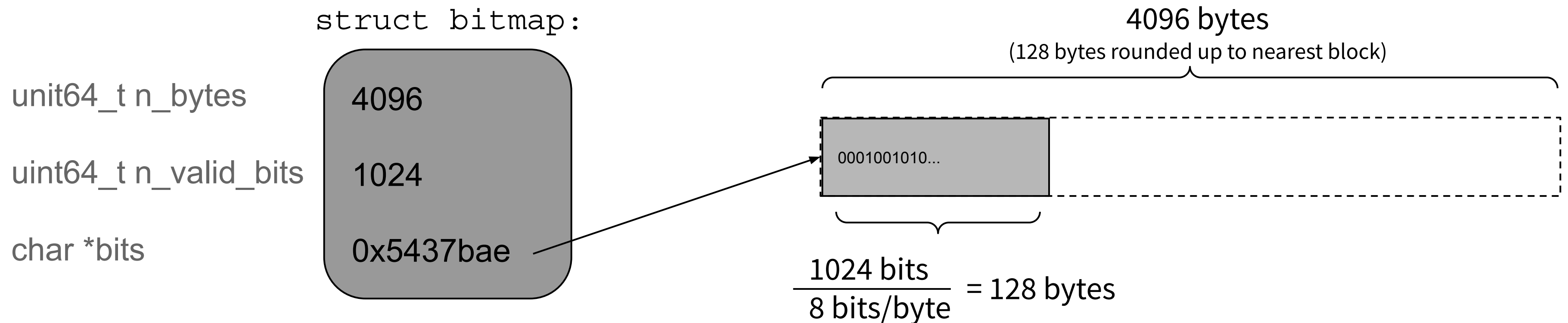
- Our FUSE implementation writes data to a file, not a disk
  - We will treat this file like a disk by reading/writing in 4096-byte blocks
    - This lets us simulate our "on disk structure" inside a 10 MiB file



Managing Allocations with Bitmaps

`bitmap.c` includes a "readable" implementation of a bitmap structure

- includes tests that show example usage
  - protected by `#ifdef BITMAP_TEST` so they are compiled out by default
- In-memory bitmap contains summary info & actual bitmap data



ReFS Metadata Structures

`refs.h` includes minimal data structure definitions for

- struct superblock
- struct inode
- struct directory_entry

Together, these structures form the basis of our file system's metadata and define on-disk format

- They may need to be adjusted as your implementation supports more features, so please make adjustments
  - e.g., `struct refs_inode` does not include a "mode", which is needed for permissions (if you choose to implement that feature)

ReFS "on-disk" Format (stored in refs_disk)

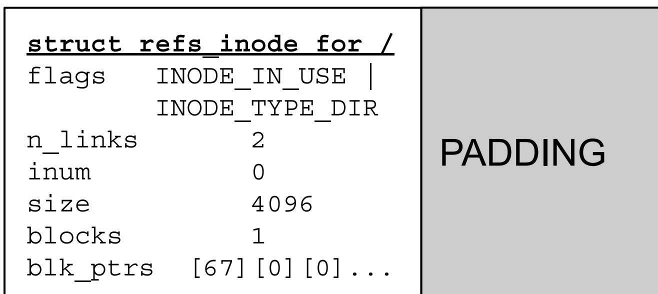
`struct refs_superblock` is located at "block 0", and it includes:

- the FS block size (4096 bytes)
- the location information for all major on-disk structures

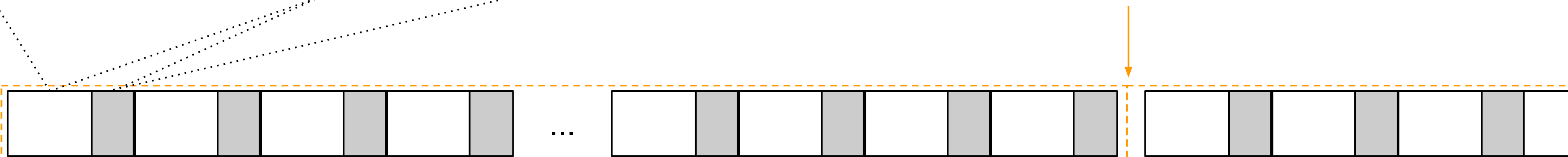
```
struct refs_superblock:
block_size          4096
num_inodes          1024
i_bitmap_start      1
i_table_start       2
num_data_blocks     2492
d_bitmap_start      66
d_region_start      67
```

"Inode table" is an array of padded inodes

- **`struct refs_inode`** stores information about a single file
- **`union inode`** exists so that we can align our inode table to block boundaries.
  - Since **`char pad[INODE_SIZE]`** is larger than **`struct refs_inode;`**, it adds padding



Note: when we read an inode, we must:
- identify which 4096-byte block that inode belongs to
- access the particular inode from the array of inodes in that block



Directories are arrays of `struct dir_entry`

- `struct dir_entry` maps an inode number to a path component
  - We use a fixed-size entry to simplify the design, but this means we need to store the length that the actual path string uses
  - We also use an "is_valid" flag to logically delete entries. this saves us from needing to shift all later array entries back by one position whenever we delete value

```
struct refs_inode for /
flags     INODE_IN_USE |
          INODE_TYPE_DIR
n_links        2
inum           0
size           4096
blocks         1
blk_ptrs  [67][0][0]...
```

inum	valid?	len	path[MAX_PATH_LEN]
0	1	3	".."
1	1	2	"."
~~47~~	~~0~~	~~8~~	~~"deleted"~~
51	1	4	"foo"
…	…	…	…

Mounting ReFS

When you mount, refs.c does one of two things:

1. If the file **`refs_disk`** exists, **`refs.c`** reads the superblock (block 0), then populates the ReFS in-memory data structures using the contents of that existing file system
2. If the file **`refs_disk`** does not exist, **`refs.c`** creates an empty 10-MiB file called **`refs_disk`**, and then creates the "/" directory
    - This involves allocating inode 0, and creating a directory data block with entries for "`.`" and "`..`"
    - The now-initialized file system is written to **`refs_disk`**, making it persistent

Summary

The starter code is there as a template

- You may modify it as you see fit, or ignore it entirely

If you use the starter code, you may need to add new fields to your data structures if necessary to implement functionality

- For example, if you want to add permissions to files, you may want to add field in the `struct inode` to store the mode