

A4: Layered Block-Structured File System

Slides originally by Prof. van Renesse
Current version by Anne Bracy

Introduction

File System

abstraction that provides persistent, *named* data

Block Store

abstraction providing access to a sequence of *numbered* blocks.
(No names.)

Physical Device
(e.g., DISK)

Disk: sectors identified with logical block addresses, specifying surface, track, and sector to be accessed.

Layered Abstractions to access storage

(*HIGHLY SIMPLIFIED* FIGURE 11.7 from book)

Block Store Abstraction

Provides a disk-like interface:

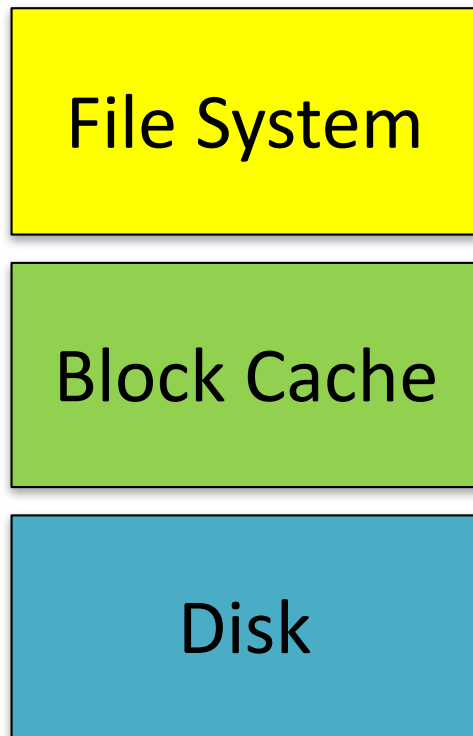
- a sequence of blocks numbered 0, 1, ... (typically a few KB)
- you can read or write 1 block at a time

<code>nblocks()</code>	returns size of the block store in #blocks
<code>read(block_num)</code>	returns contents of given block number
<code>write(block_num, block)</code>	writes block contents at given block num
<code>setsize(size)</code>	sets the size of the block store

A4 has you work with
multiple versions / instantiations of
this abstraction.

1st Instantiation: Block Cache

All requests for a given block go through block cache



- Benefit #1: Performance
 - Caches recently read blocks
 - Buffers recently written blocks (to be written later)
- Benefit #2: Synchronization:
 - For each entry, OS adds information to:
 - prevent a process from reading block while another writes
 - ensure that a given block is only fetched from storage device once, even if it is simultaneously read by many processes

Heads up about the code!

This entire code base is what happens when you want object oriented programming, but you only have C.

Put on your C++ / Java Goggles!

`block_store_t` (a block store type)
is essentially an abstract class

Contents of block_store.h

```
#define BLOCK_SIZE      512      // # bytes in a block
```

```
typedef unsigned int block_no; // index of a block
```

```
typedef struct block {  
    char bytes[BLOCK_SIZE];  
} block_t;
```

```
typedef struct block_store {  
    void *state;  
    int (*nblocks)(struct block_store *this_bs);  
    int (*read)(struct block_store *this_bs, block_no offset, block_t *block);  
    int (*write)(struct block_store *this_bs, block_no offset, block_t *block);  
    int (*setsize)(struct block_store *this_bs, block_no size);  
    void (*destroy)(struct block_store *this_bs);  
} block_store_t;
```

← function pointers,
AKA class methods

← *poor man's class*

None of this is data! All typedefs!

Block Store Instructions

- `block_store_t *xxx_init(...)` ← “*constructor*”
 - Name & signature varies, sets up the fn pointers
- `int nblocks(...)`
- `read(...)`
- `write(...)`
- `setsize(...)`
- `destroy()` ← “*destructor*”
 - frees everything associated with this block store

Sample Program: just a Disk

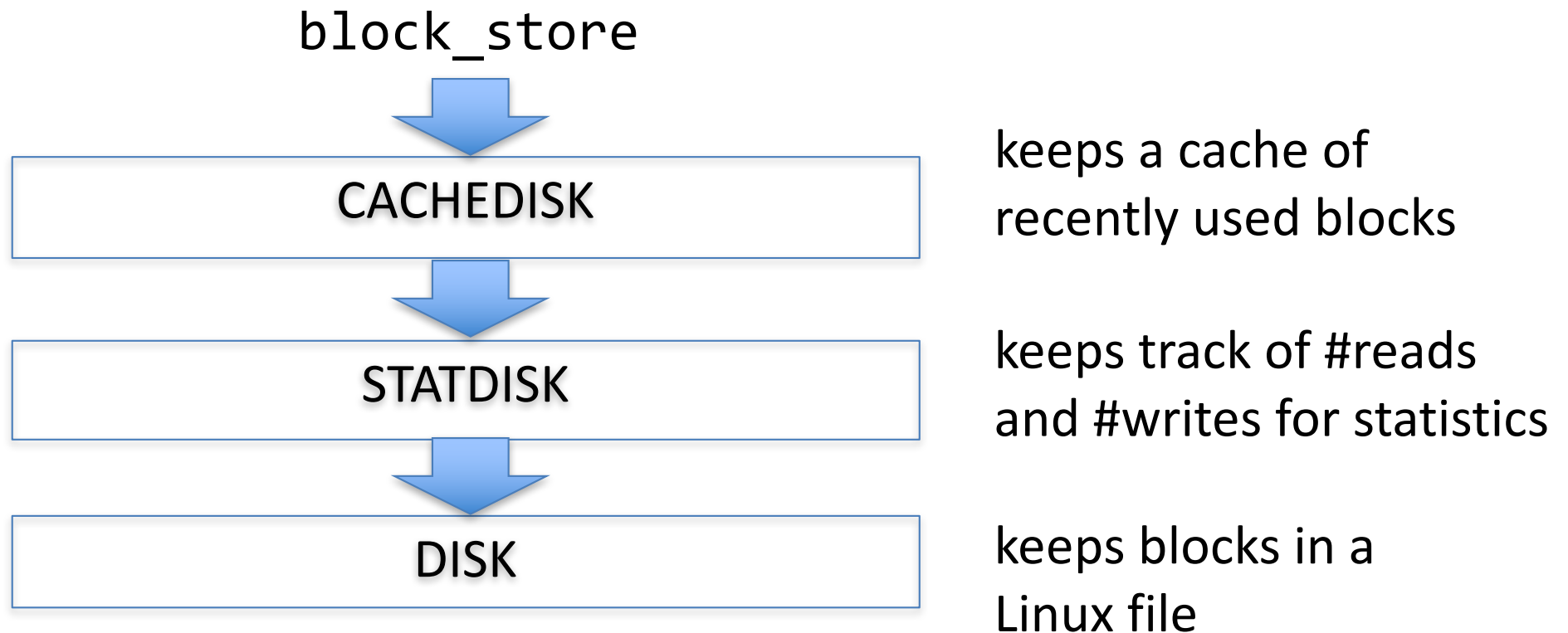
```
#include ...
#include "block_store.h"

int main(){
    block_store_t *disk = disk_init("disk.dev", 1024);
    block_t block;
    strcpy(block.bytes, "Hello World");
    (*disk->write)(disk, 0, &block);
    (*disk->destroy)(disk);
    return 0;
}
```

```
gcc -g block_store.c sample.c
gdb then check out disk.dev
```


Block Stores can be Layered!

Each layer presents a block store abstraction



Example code with layers

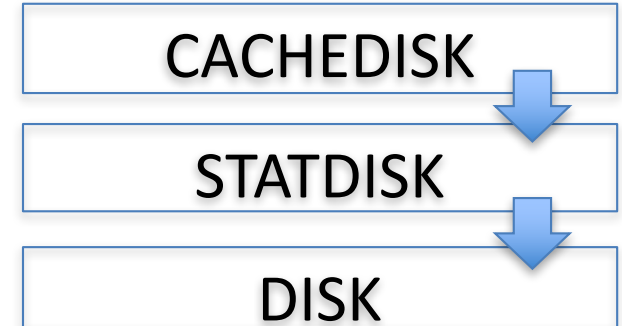
```
#define CACHE_SIZE 10 // #blocks in cache

block_t cache[CACHE_SIZE];

int main(){
    block_store_t *disk = disk_init("disk.dev", 1024);
    block_store_t *sdisk = statdisk_init(disk);
    block_store_t *cdisk = cachedisk_init(sdisk, cache, CACHE_SIZE);

    block_t block;
    strcpy(block.bytes, "Hello World");
    (*cdisk->write)(cdisk, 0, &block);
    (*cdisk->destroy)(cdisk);
    (*sdisk->destroy)(sdisk);
    (*disk->destroy)(disk);

    return 0;
}
```



```
gcc -g block_store.c statdisk.c cachedisk.c layer.c
```

Example Layers

```
block_store_t *statdisk_init(block_store_t *below);  
    // counts all reads and writes  
  
block_store_t *debugdisk_init(block_store_t *below, char *descr);  
    // prints all reads and writes  
  
block_store_t *checkdisk_init(block_store_t *below);  
    // checks that what's read is what was written  
  
block_store_t *disk_init(char *filename, int nblocks)  
    // simulated disk stored on a Linux file  
    // (could also use real disk using /dev/*disk devices)  
  
block_store_t *ramdisk_init(block_t *blocks, nblocks)  
    // a simulated disk in memory, fast but volatile
```

How to write a layer

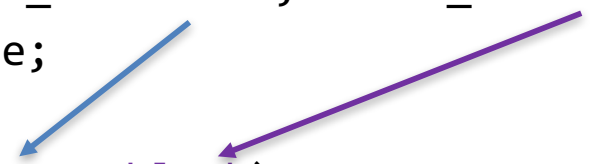
```
struct statdisk_state {
    block_store_t *below;           // block store below
    unsigned int nread, nwrite;     // stats
};

block_store_t *statdisk_init(block_store_t *below){
    struct statdisk_state *sds = calloc(1, sizeof(*sds));
    sds->below = below;

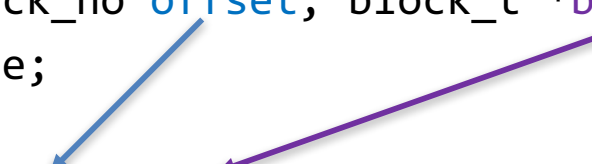
    block_store_t *this_bs = calloc(1, sizeof(*this_bs));
    this_bs->state = sds;
    this_bs->nblocks = statdisk_nblocks;
    this_bs->setsize = statdisk_setsize;
    this_bs->read = statdisk_read;
    this_bs->write = statdisk_write;
    this_bs->destroy = statdisk_destroy;
    return this_bs;
}
```

statdisk implementation (cont'd)

```
int statdisk_read(block_store_t *this_bs, block_no offset, block_t *block){
    struct statdisk_state *sds = this_bs->state;
    sds->nread++;
    return (*sds->below->read)(sds->below, offset, block);
}
```



```
int statdisk_write(block_store_t *this_bs, block_no offset, block_t *block){
    struct statdisk_state *sds = this_bs->state;
    sds->nwrite++;
    return (*sds->below->write)(sds->below, offset, block);
}
```

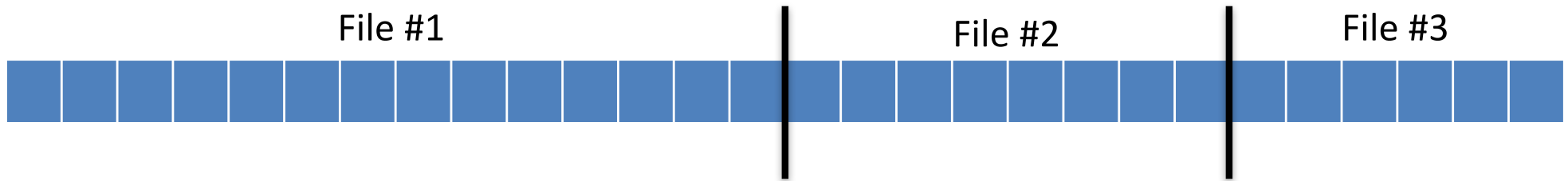


```
void statdisk_destroy(block_store_t *this_bs){
    free(this_bs->state);
    free(this_bs);
}
```

Why don't we destroy the below?

One File → Many Files

- One could create multiple partitions, one for each file, but that has very similar problems to partitioning physical memory among processes

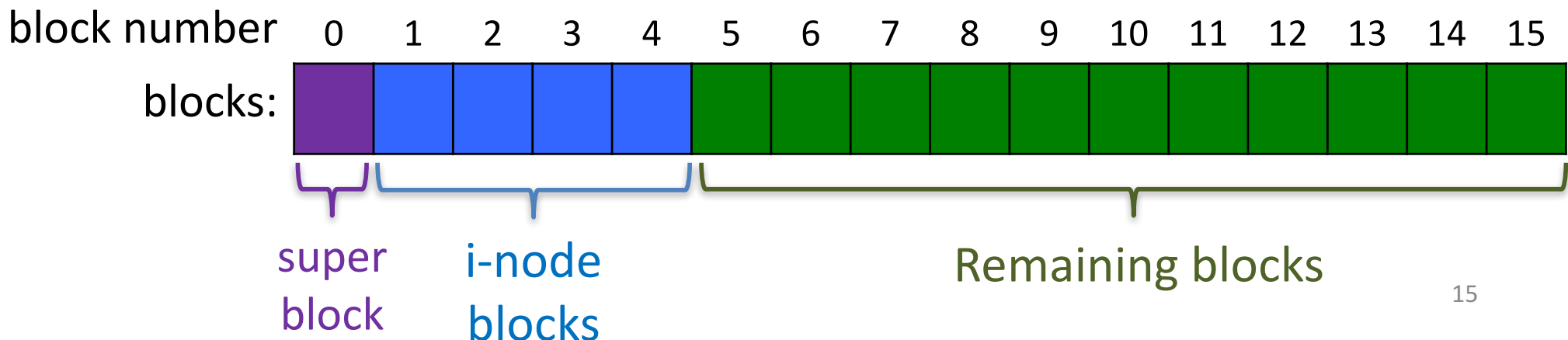


- You want something similar to paging
 - more efficient and flexible sharing
 - techniques are very similar!

Solution: File Systems!

Treedisk

- A file system, similar to Unix file systems (*in 2 weeks*)
- Initialized to support N virtual block stores (AKA files)
- Underlying block store (below) partitioned into 3 sections:
 1. **Superblock:** block #0
 2. Fixed number of *i-node blocks*: starts at block #1
 - Function of N (enough to store N i-nodes)
 3. **Remaining blocks:** starts after i-node blocks
 - *data blocks, free blocks, indirect blocks, freelist blocks*

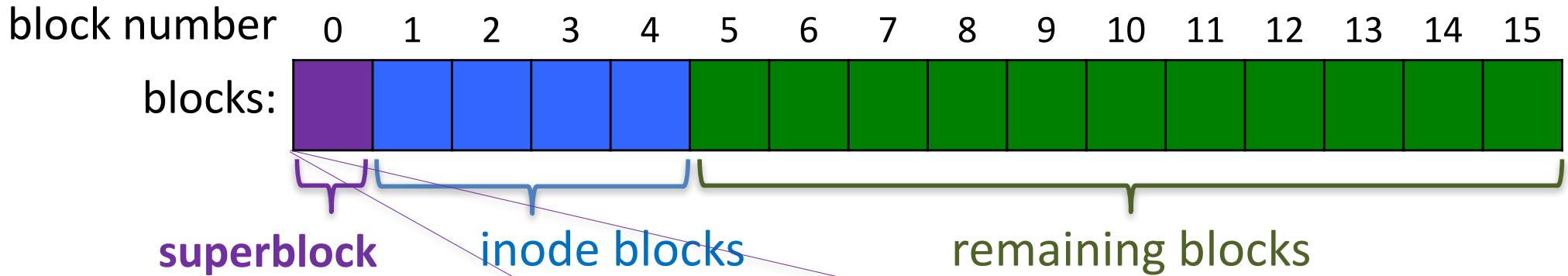


Types of Blocks in Treedisk

```
union treedisk_block {  
    block_t datablock;  
    struct treedisk_superblock superblock;  
    struct treedisk_inodeblock inodeblock;  
    struct treedisk_freelistblock freelistblock;  
    struct treedisk_indirblock indirblock;  
};
```

- Superblock: the 0th block below
- Freelistblock: list of all unused blocks below
- I-nodeblock: list of inodes
- Indirblock: list of blocks
- Datablock: just data

treedisk Superblock



// one per underlying block store

```
struct treedisk_superblock {
```

```
    block_no n_inodeblocks;
```

```
    block_no free_list;
```

```
    // 1st block on free list
```

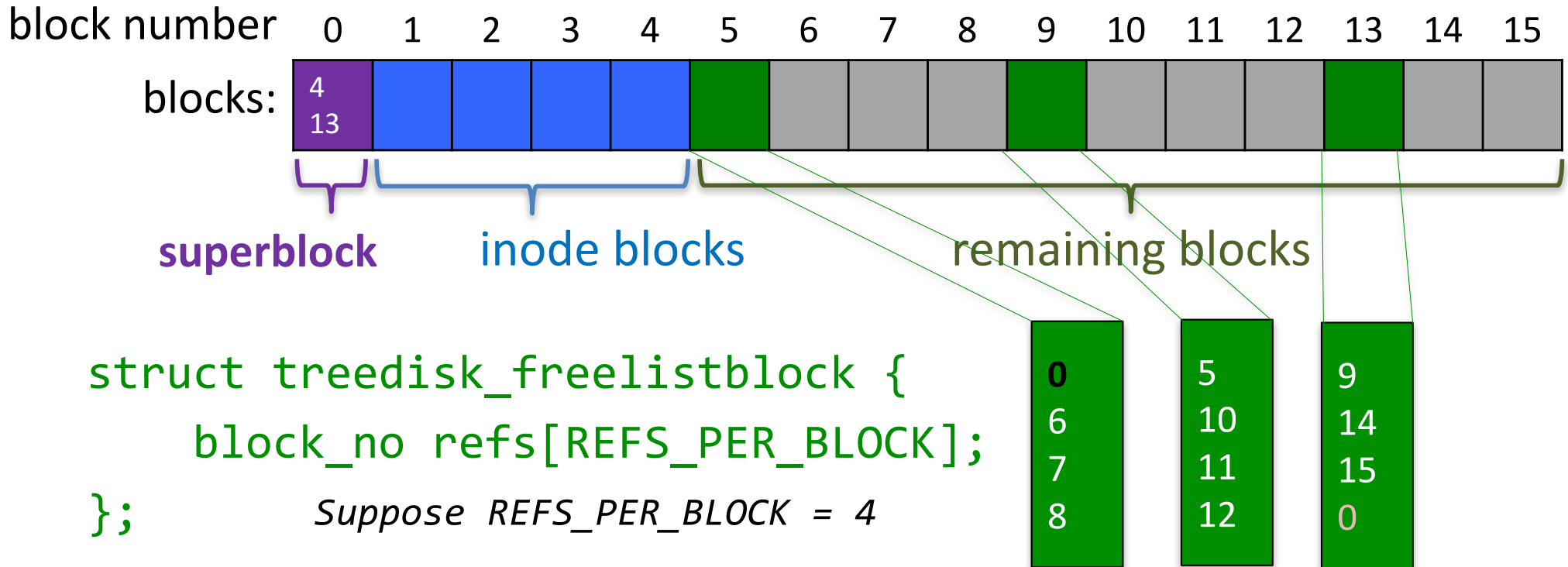
```
    // 0 means no free blocks
```

```
};
```

```
n_inodeblocks 4  
free_list      ?  
(some green box)
```

Notice: there are no pointers. Everything is a block number.

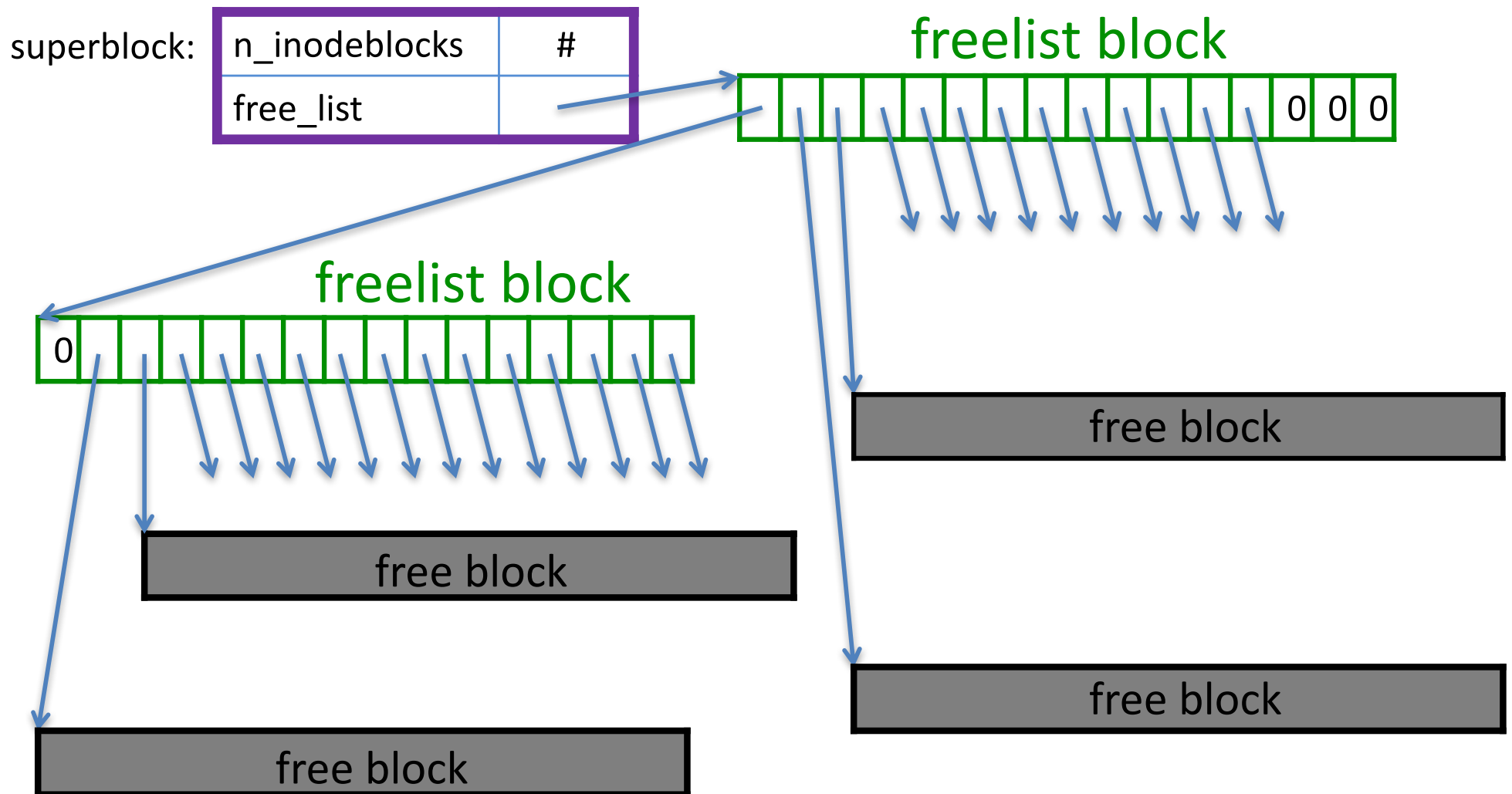
treedisk Free List



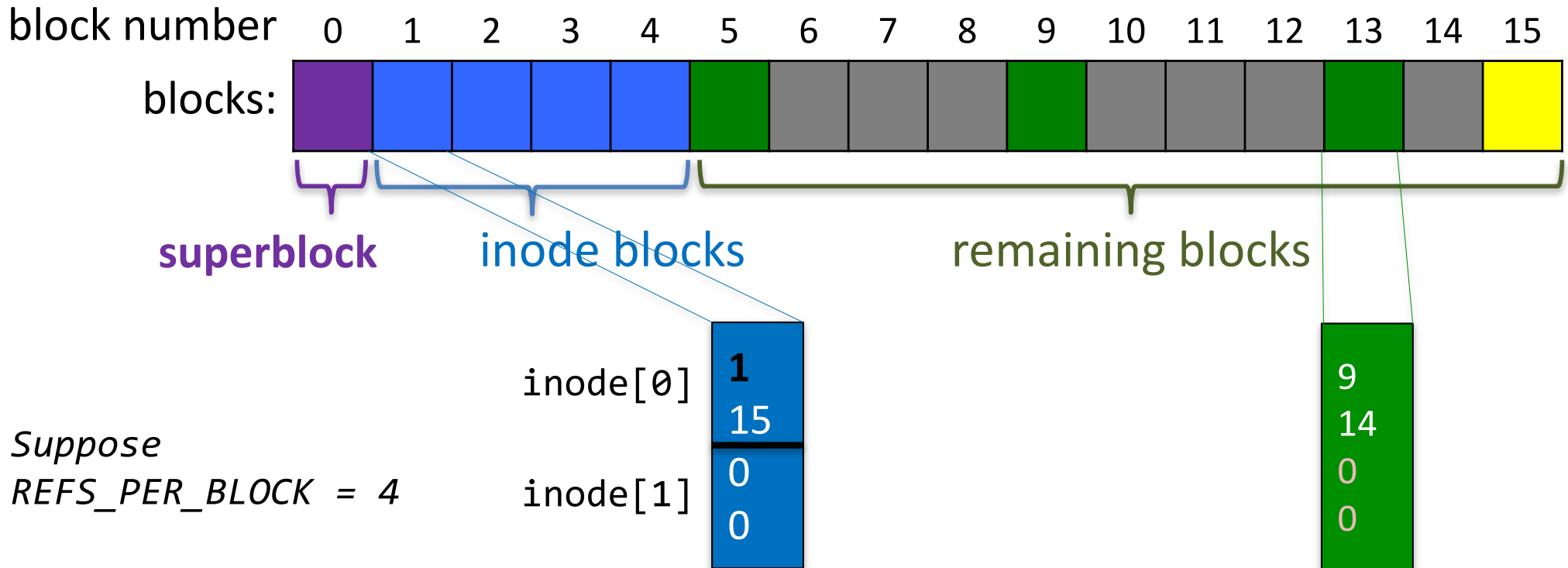
refs[0]: # of another freelistblock or **0 if end of list**

refs[i]: # of free block for $i > 1$, **0 if slot empty**

treedisk free list



treedisk I-node block



Suppose

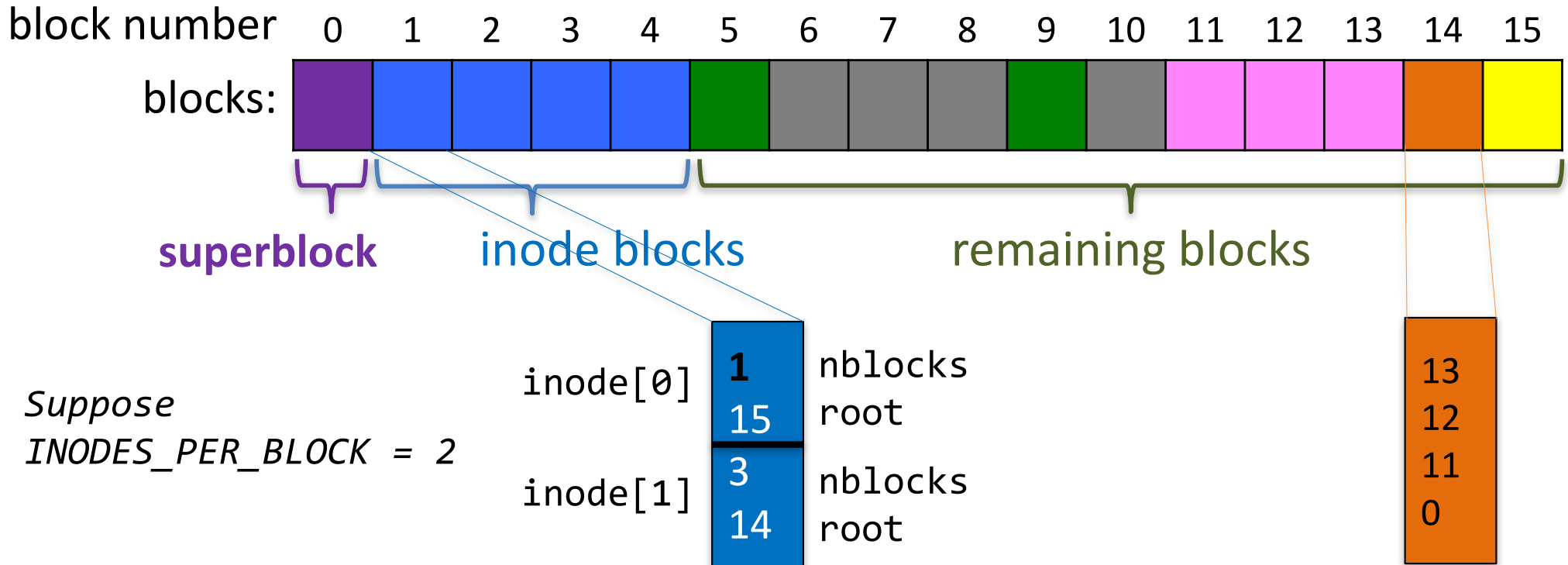
`REFS_PER_BLOCK = 4`

```
struct treedisk_inodeblock {
    struct treedisk_inode inodes[INODES_PER_BLOCK];
};
```

What if the file is bigger than 1 block?

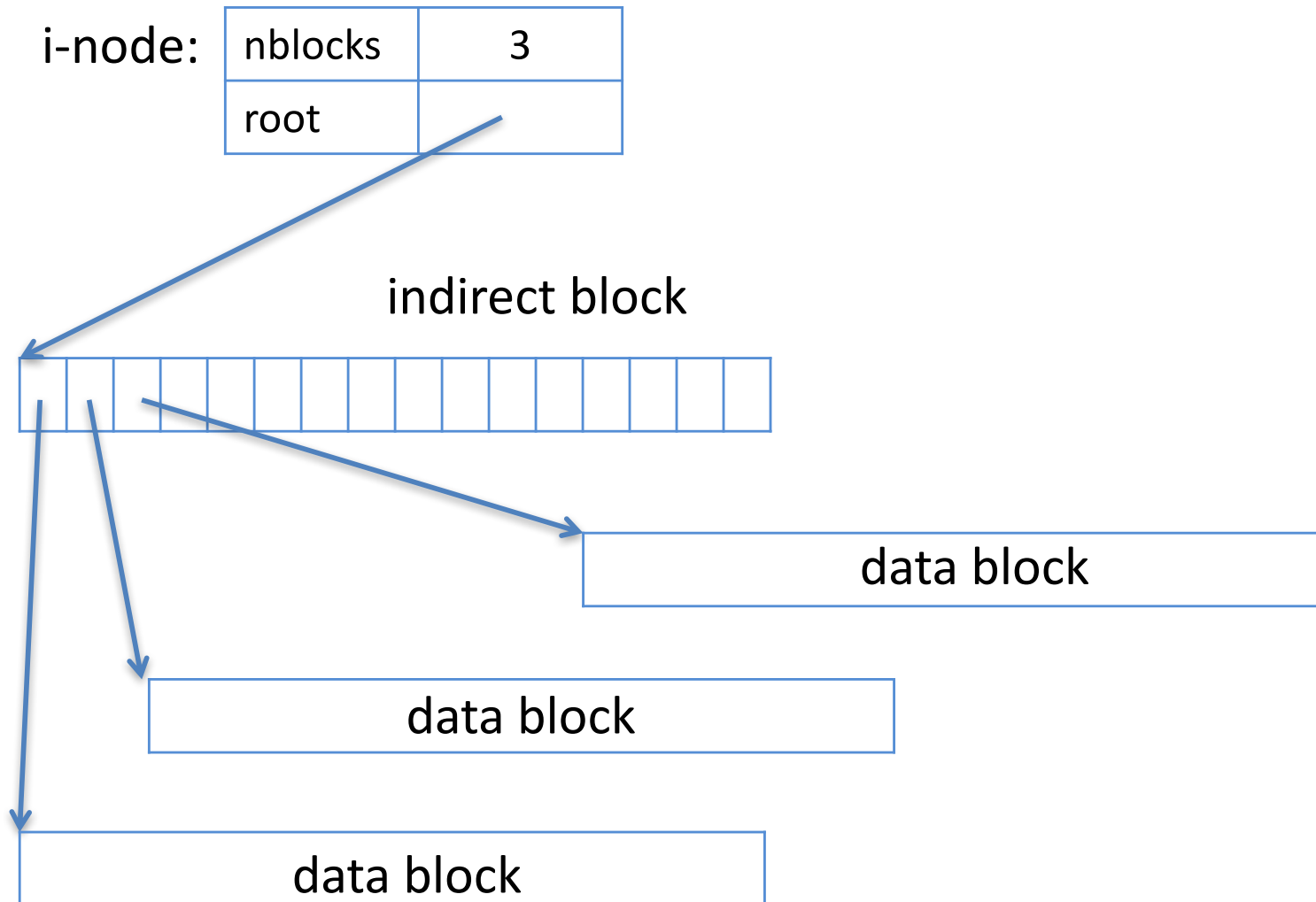
```
struct treedisk_inode {
    block_no nblocks;    // # blocks in virtual block store
    block_no root;      // block # of root node of tree (or 0)
};
```

treedisk Indirect block



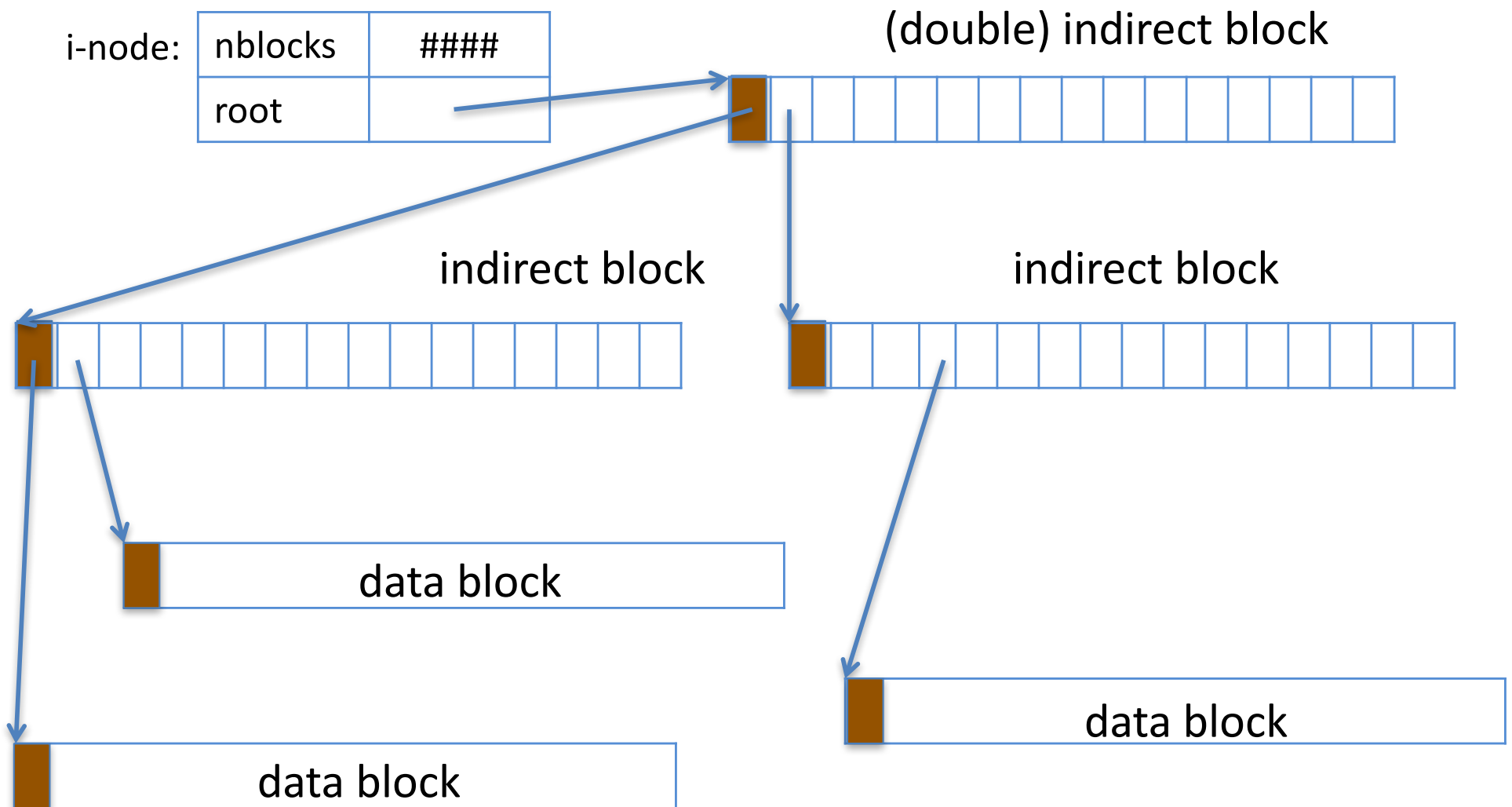
```
struct treedisk_indirblock {
    block_no refs[REFS_PER_BLOCK];
};
```

virtual block store: 3 blocks



What if the file is bigger than 3 blocks?

treedisk virtual block store



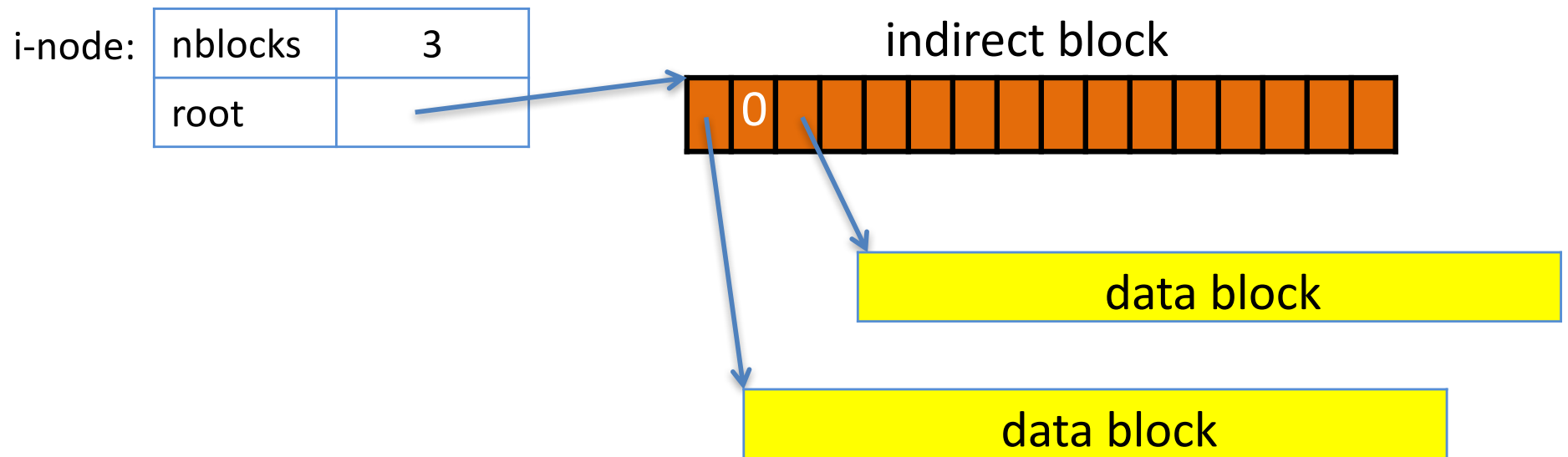
How do I know if this is data or a block number?

treedisk virtual block store

- all data blocks at bottom level
- #levels: $\text{ceil}(\log_{\text{RPB}}(\#\text{blocks})) + 1$
RPB = REFS_PER_BLOCK
- For example, if rpb = 16:

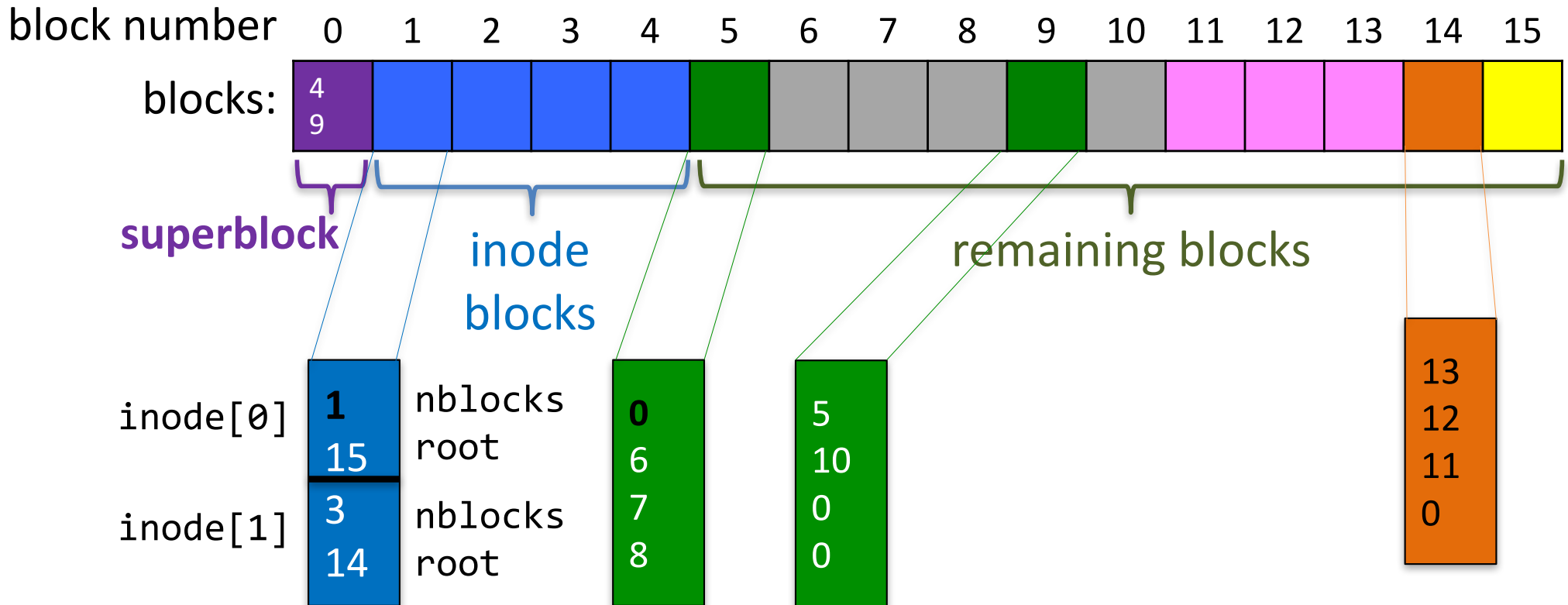
#blocks	#levels
0	0
1	1
2 - 16	2
17 - 256	3
257 - 4096	4

virtual block store: with hole



- Hole appears as a virtual block filled with null bytes
- pointer to indirect block can be 0 too
- virtual block store can be much larger than the “physical” block store underneath!

Putting it all together



A short-lived treedisk file system

```
#define DISK_SIZE 1024
#define MAX_INODES 128

int main(){
    block_store_t *disk = disk_init("disk.dev", DISK_SIZE);

    treedisk_create(disk, MAX_INODES);

    treedisk_check(disk); // optional: check integrity of file system

    (*disk->destroy)(cdisk);

    return 0;
}
```

Example code with treedisk

```
block_t cache[CACHE_SIZE];

int main(){
    block_store_t *disk = disk_init("disk.dev", 1024);
    block_store_t *cdisk = cachedisk_init(disk, cache, CACHE_SIZE);
    treedisk_create(disk, MAX_INODES);
    block_store_t *disk0 = treedisk_init(cdisk, 0);
    block_store_t *disk1 = treedisk_init(cdisk, 1);

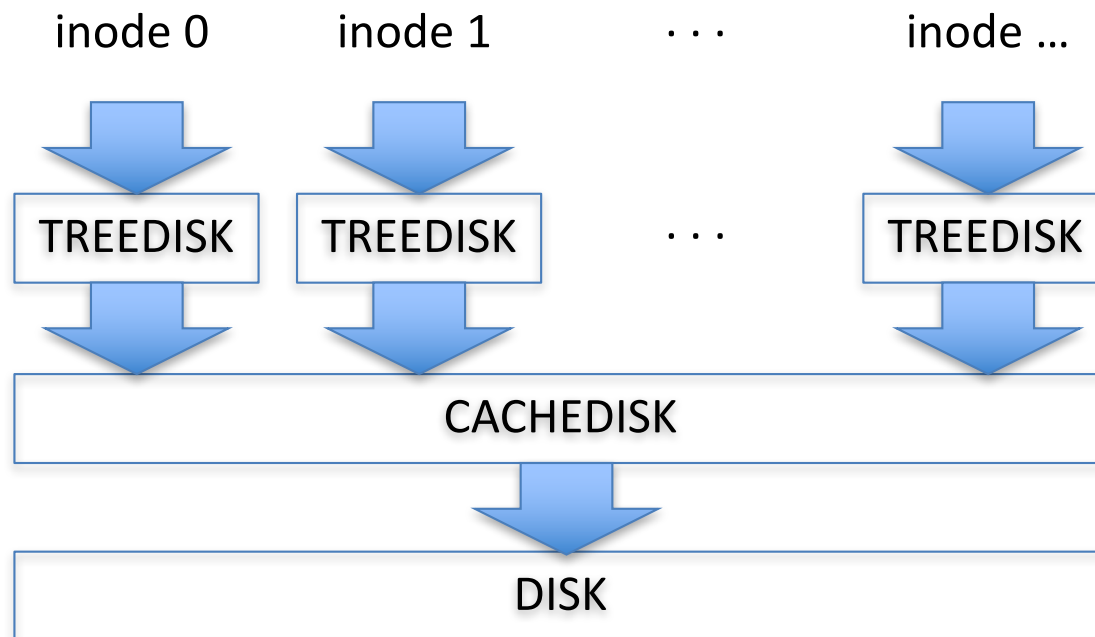
    block_t block;
    (*disk0->read)(disk0, 4, &block);
    (*disk1->read)(disk1, 4, &block);

    (*disk0->destroy)(disk0);
    (*disk1->destroy)(disk1);
    (*cdisk->destroy)(cdisk);
    (*disk->destroy)(cdisk);

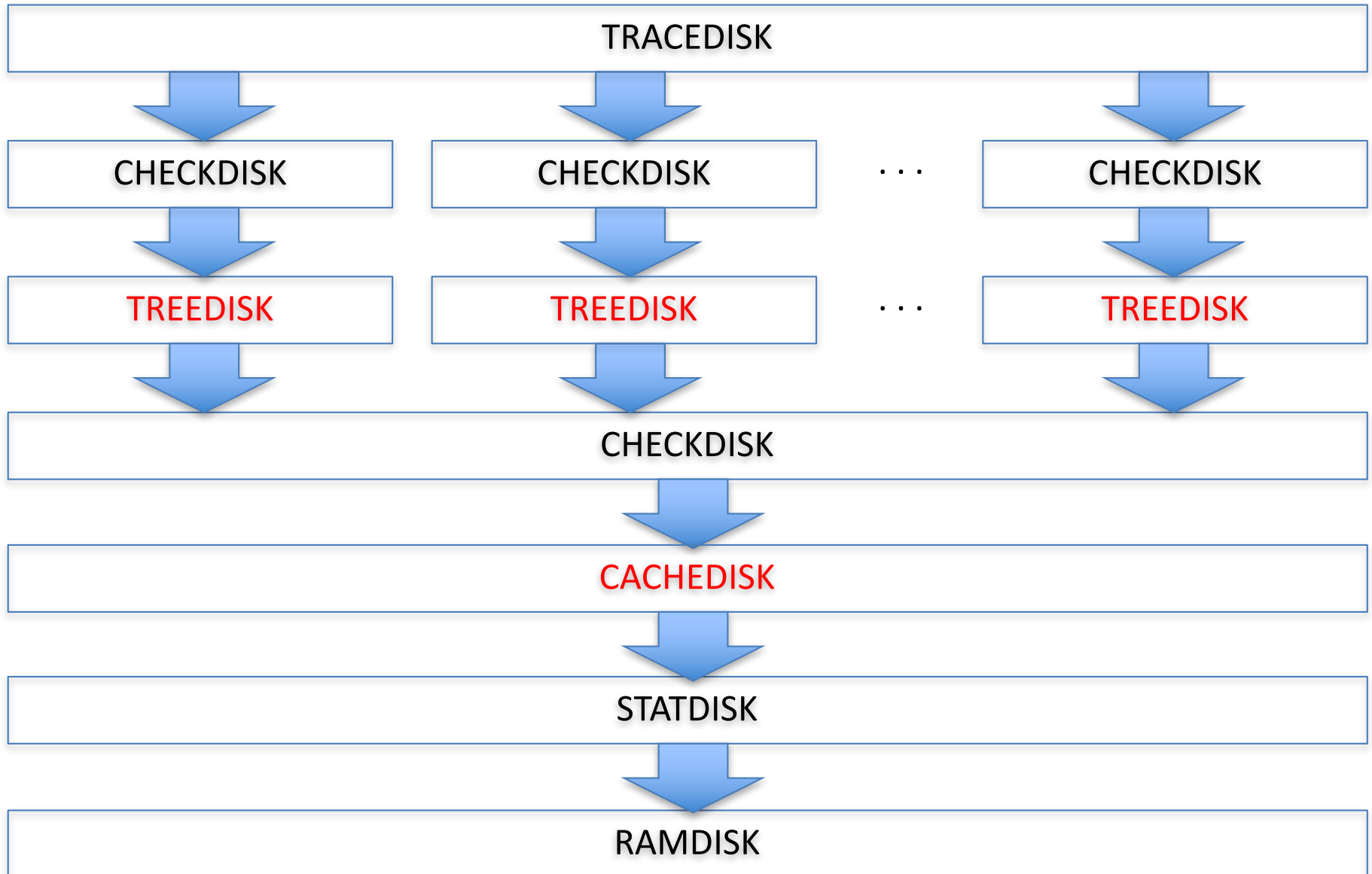
    return 0;
}
```

Layering on top of treedisk

```
block_store_t *treedisk_init(block_store_t *below,  
                             unsigned int inode_no);
```



trace utility



tracedisk

- ramdisk is bottom-level block store
- tracedisk is a top-level block store
 - or “application-level” if you will
 - you can’t layer on top of it

```
block_store_t *tracedisk_init(  
    block_store_t *below,  
    char *trace,           // trace file name  
    unsigned int n_inodes);
```

Example trace file

```
W:0:0 // write inode 0, block 0
N:0:1 // checks if inode 0 is of size 1
W:1:1 // write inode 1, block 1
N:1:2 // checks if inode 1 is of size 2
R:1:1 // read inode 1, block 1
S:1:0 // set size of inode 1 to 0
N:1:0 // checks if inode 0 is of size 0
```

if N fails, prints “!!CHKSIZE ..”

Compiling and Running

- run “make” in the release directory
 - this generates an executable called “trace”
- run “./trace”
 - this reads trace file “trace.txt”
 - you can pass another trace file as argument
 - ./trace myowntracefile

Output to be expected

```
$ make
cc -Wall -c -o trace.o trace.c
. . .
cc -Wall -c -o treedisk_chk.o treedisk_chk.c
cc -o trace trace.o block_store.o cachedisk.o checkdisk.o
debugdisk.o ramdisk.o statdisk.o tracedisk.o treedisk.o
treedisk_chk.o
```

```
$ ./trace
```

blocksize: 512	Trace
refs/block: 128	W:0:0
!!TDERR: setsize not yet supported	N:0:1
!!ERROR: tracedisk_run: setsize(1, 0) failed	W:0:1
!!CHKSIZE 10: nblocks 1: 0 != 2	N:0:2
!\$STAT: #nblocks: 5	W:1:0
!\$STAT: #nsetsize: 0	N:1:1
!\$STAT: #nread: 32	W:1:1
!\$STAT: #nwrite: 20	N:1:2

Cmd:inode:block 34

A4: Part 1/3

Implement `treedisk_setsize(0)`

- currently it generates an error
- what you need to do:
 - iterate through all the blocks in the inode
 - put them on the free list

Useful functions:

- `treedisk_get_snapshot`

A4: Part 2/3

Implement cachedisk

- currently it doesn't actually do anything
- what you need to do:
 - pick a caching algorithm: LRU, MFU, or design your own
 - go wild!
 - implement it within `cachedisk.c`
 - *write-through cache!!*
 - consult the web for caching algorithms!

A4: Part 3/3

Implement your own trace file

- read, write, setsize operations
- at most 10,000 commands
- at most 128 inodes
- at most $1 \ll 27$ block size
- try to make it really hard for a caching layer to be effective
 - e.g., random reads / writes

What to submit

- `treedisk.c` // with `treedisk_setsize()`
- `cachedisk.c`
- `trace.txt`

The **Big Red** Caching Contest!!!

- We will run everybody's trace against everybody's treedisk and cachedisk
- We will run this on top of a statdisk
- We will count the total number of read operations
- The winner is whomever ends up doing the fewest read operations to the underlying disk
- Does not count towards grade of A4, but you may win fame and glory