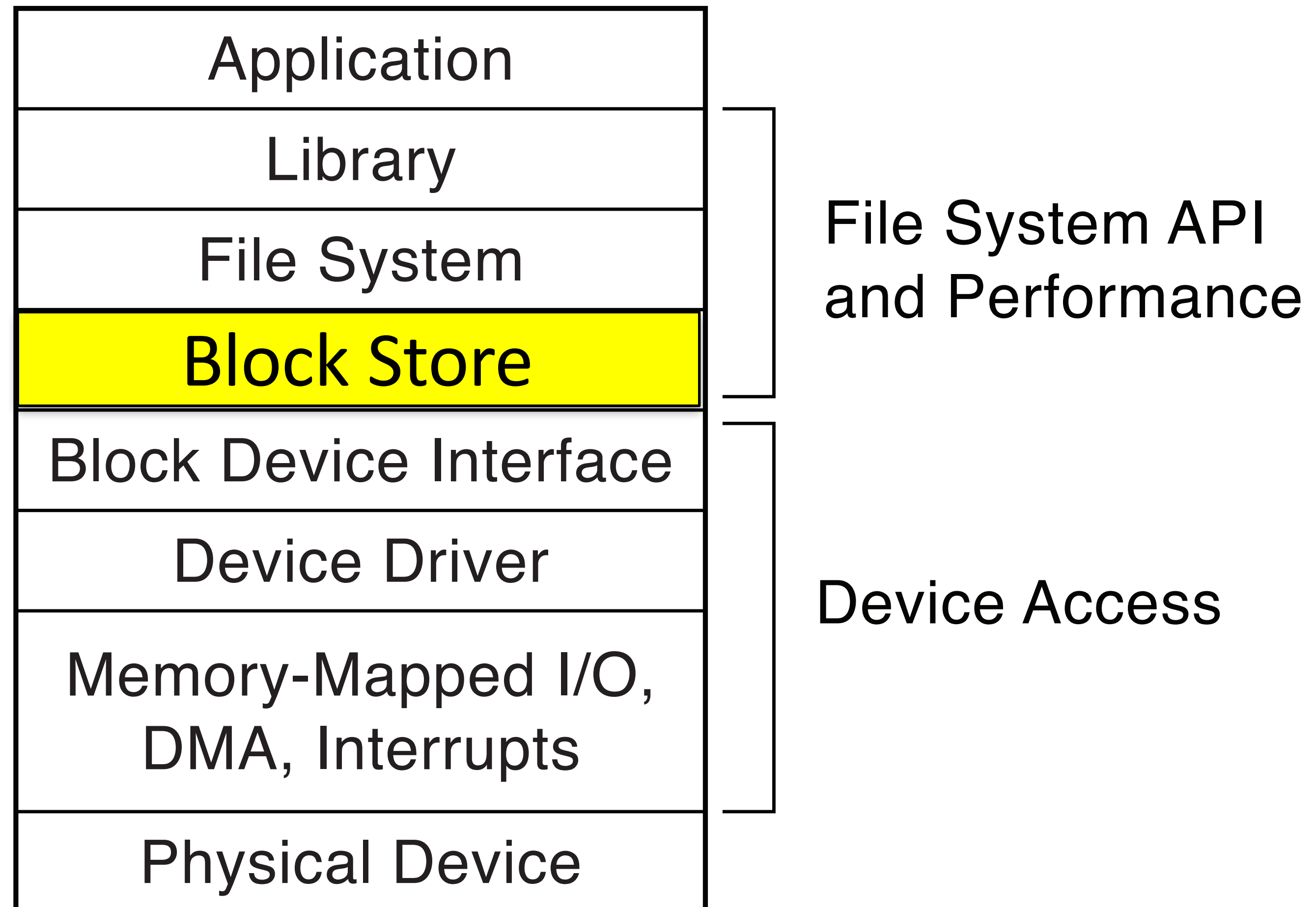
10-P4: Layered Block-Structured File System

Slides originally by Prof. van Renesse

Current version by Anne Bracy

Intro

- Underneath any file system, database system, *etc.* is more *block stores*
- Block store abstraction doesn't deal with file naming



Block Store Abstraction

- Provides a disk-like interface:
  - a sequence of blocks numbered 0, 1, …

    (typically a few kilobytes)
  - you can read or write 1 block at a time

10-P4 has you work with

multiple versions/ instantiations of this abstraction.

Block Store Benefits

- Performance:
  - Caches recently read blocks
  - Buffers recently written blocks (to be written later)

- Synchronization:
  - all requests for a given block go through block cache
  - For each entry, OS includes 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_if (a block interface)

is essentially an abstract class

Contents of block_if.h

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

typedef unsigned int block_no;  // index of a block

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

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

← *pointer to the interface*

← *poor man's class*

← *function pointers, AKA class methods*

None of this is data! All typedefs!

block_if: Block Store Interface

- xxx_init(…) → block_if ← *"constructor"*
  - Name & signature varies, sets up all the fn pointers
- nblocks() → integer
  - returns size of the block store in #blocks
- read(block number) → block
  - returns the contents of the given block number
- write(block number, block)
  - writes the block contents at the given block number
- setsize(nblocks)
  - sets the size of the block store
- destroy() ← *"destructor"*
  - frees everything associated with this block store

Simple block stores

- **disk**: simulated disk stored on a Linux file
  - block_if bif = disk_init(char *filename, int nblocks)

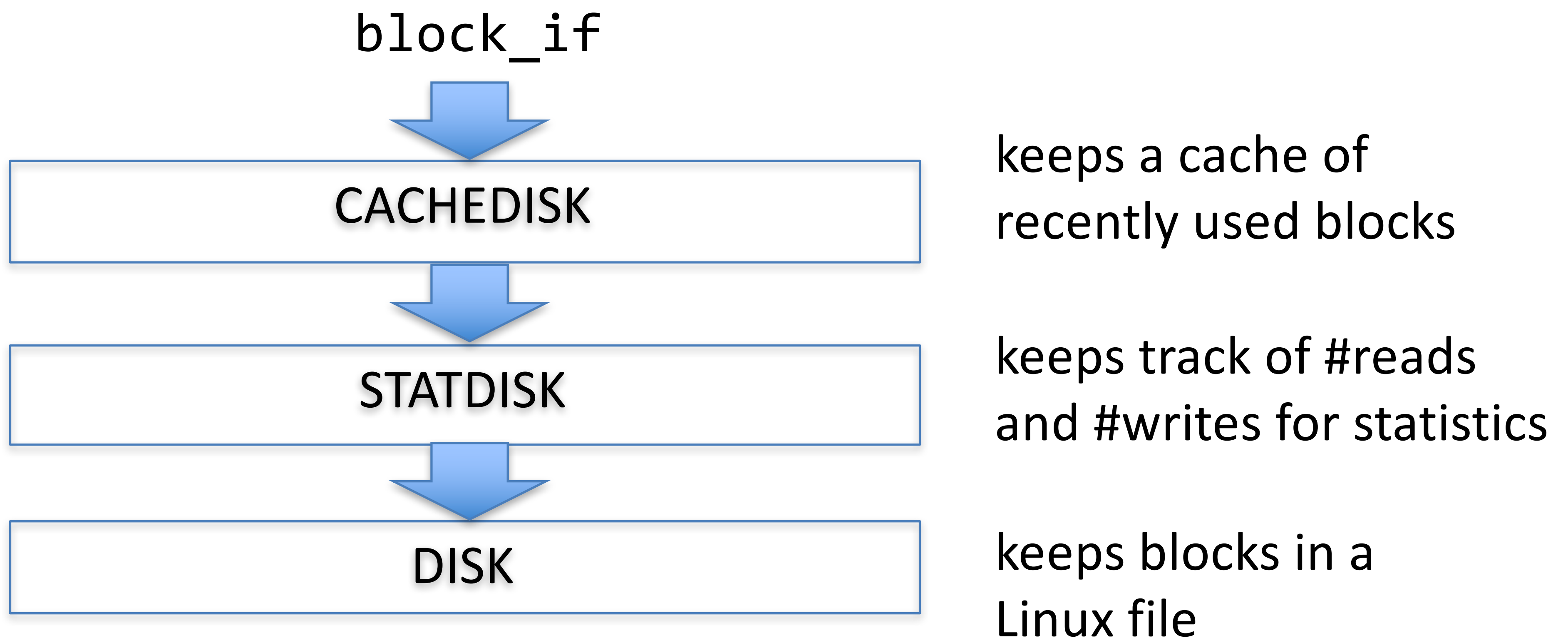
    (could also use real disk using /dev/*disk devices)
- **ramdisk**: a simulated disk in memory
  - block_if bif = ramdisk_init(block_t *blocks, nblocks)
    - Fast but volatile

Sample Program

```
#include ...
#include “block_if.h”

int main(){
  block_if 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_if.c sample.c
gdb then check out disk.dev
```

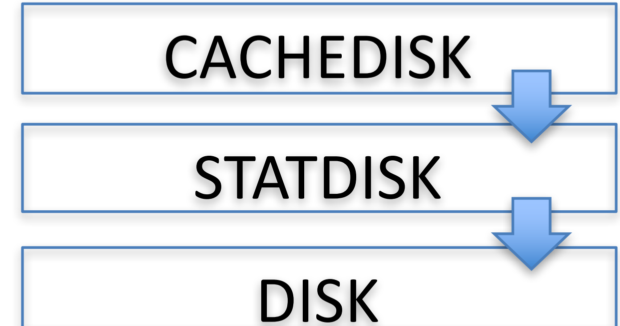
Block Stores can be Layered!

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_if disk = disk_init("disk.dev", 1024);
    block_if sdisk = statdisk_init(disk);
    block_if 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_if.c statdisk.c cachedisk.c layer.c
```

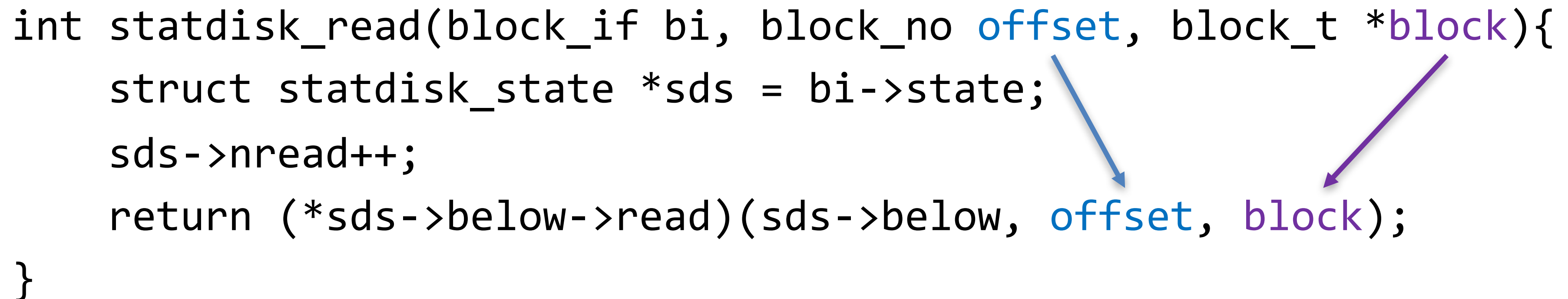
Example Layers

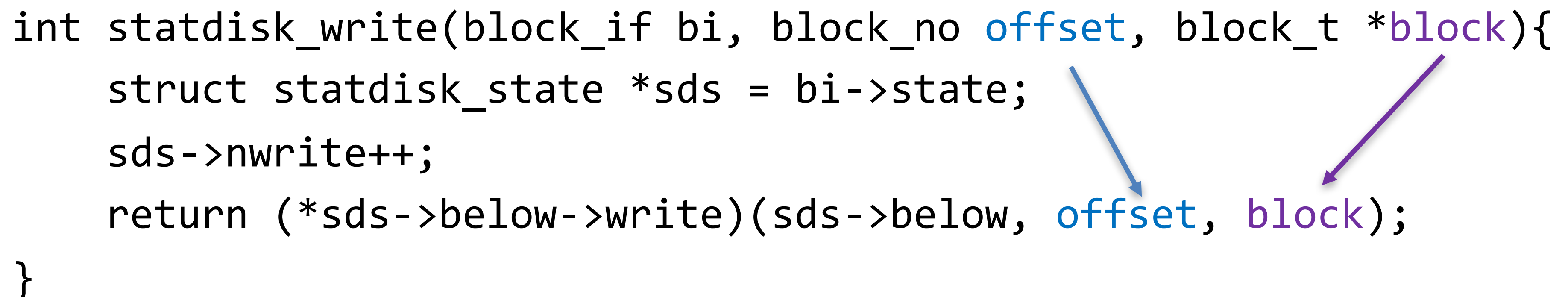
```
block_if clockdisk_init(block_if below,
                  block_t *blocks, block_no nblocks);
    // implements CLOCK cache allocation / eviction

block_if statdisk_init(block_if below);
    // counts all reads and writes

block_if debugdisk_init(block_if below, char *descr);
    // prints all reads and writes

block_if checkdisk_init(block_if below);
    // checks that what's read is what was written
```

How to write a layer

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

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

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

statdisk implementation, cont'd





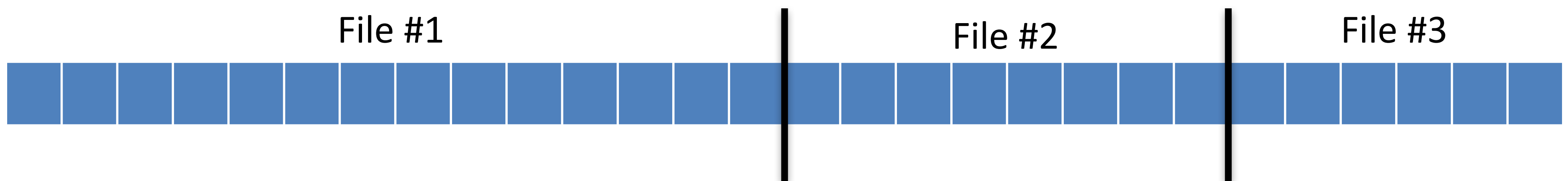

```
void statdisk_destroy(block_if bi){
    free(bi->state);
    free(bi);
}
```

Why don't we destroy the below?

all 3 functions declared static

Sharing a Block Store

- 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!

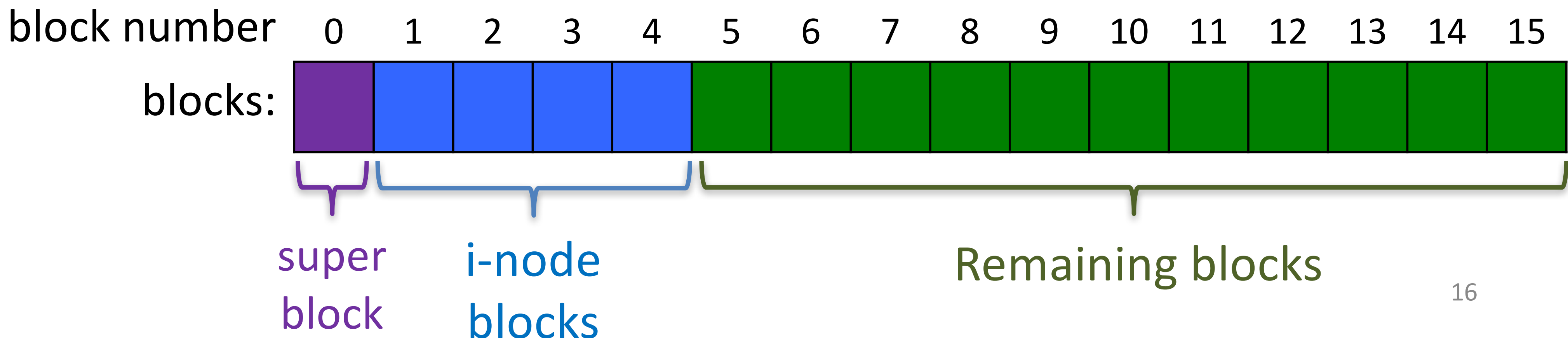


Solution: File Systems!

Treedisk

- A file system, similar to Unix file systems (*this Thursday*)
- 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*

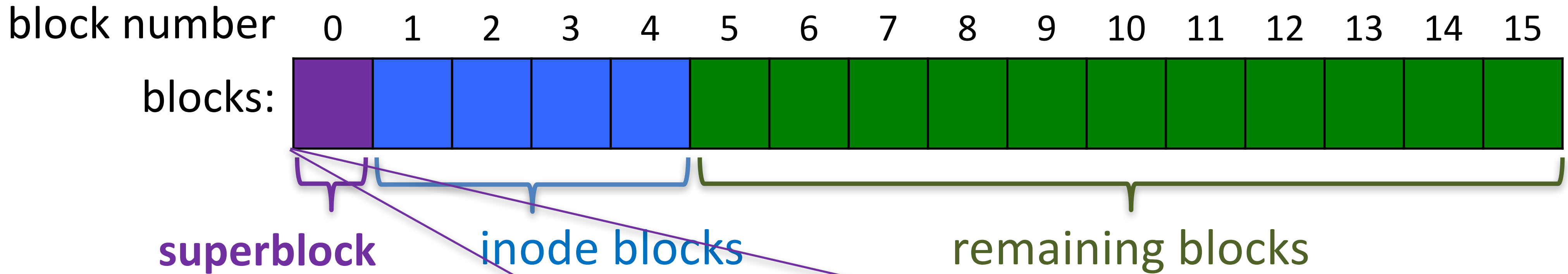


Types of Blocks in Treedisk

```
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

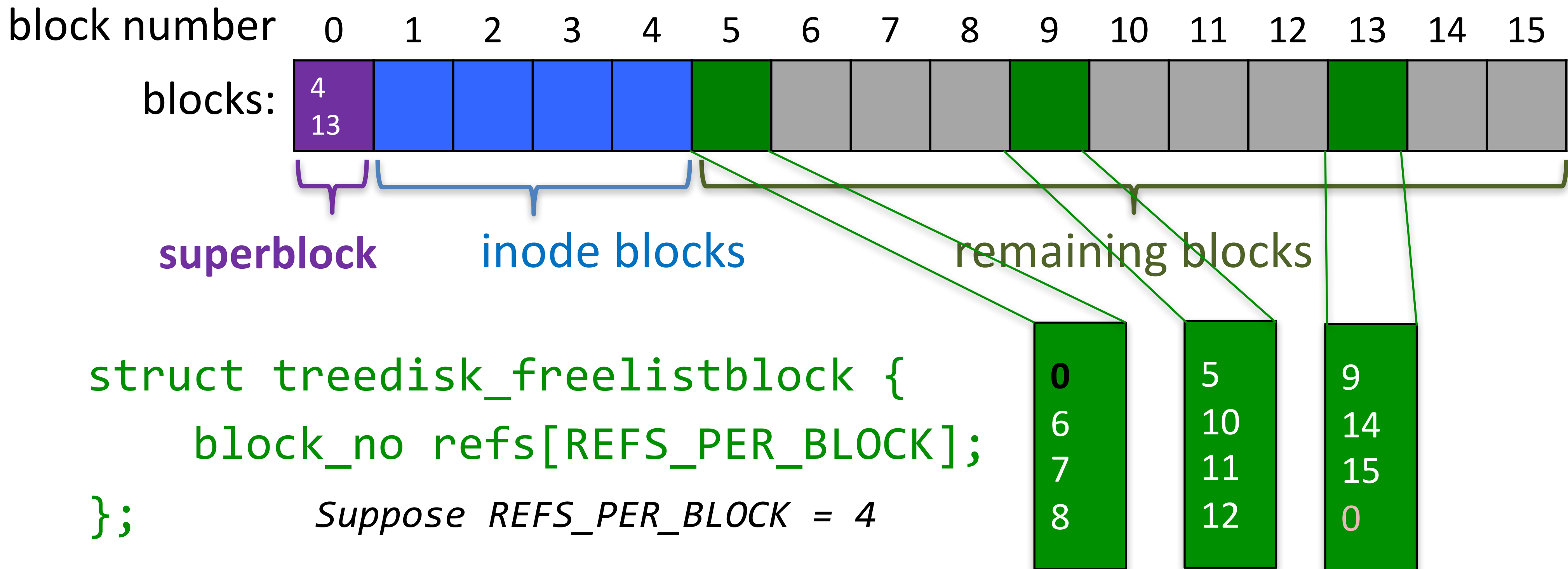


n_inodeblocks 4
free_list ?
(some green box)

```
// 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
};
```

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

treedisk Free List



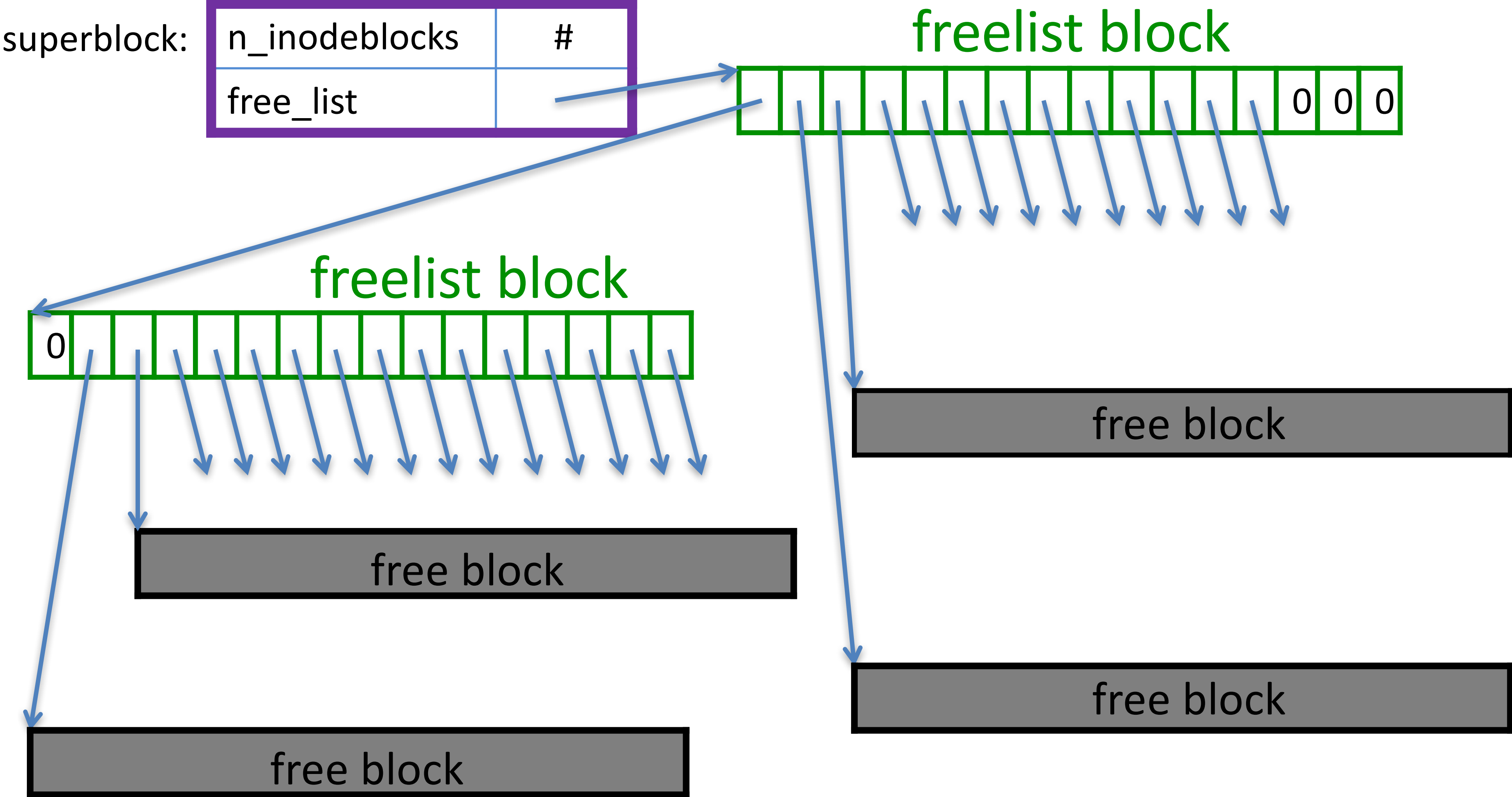

```
struct treedisk_freelistblock {
    block_no refs[REFS_PER_BLOCK];
};
```

Suppose REFS_PER_BLOCK = 4

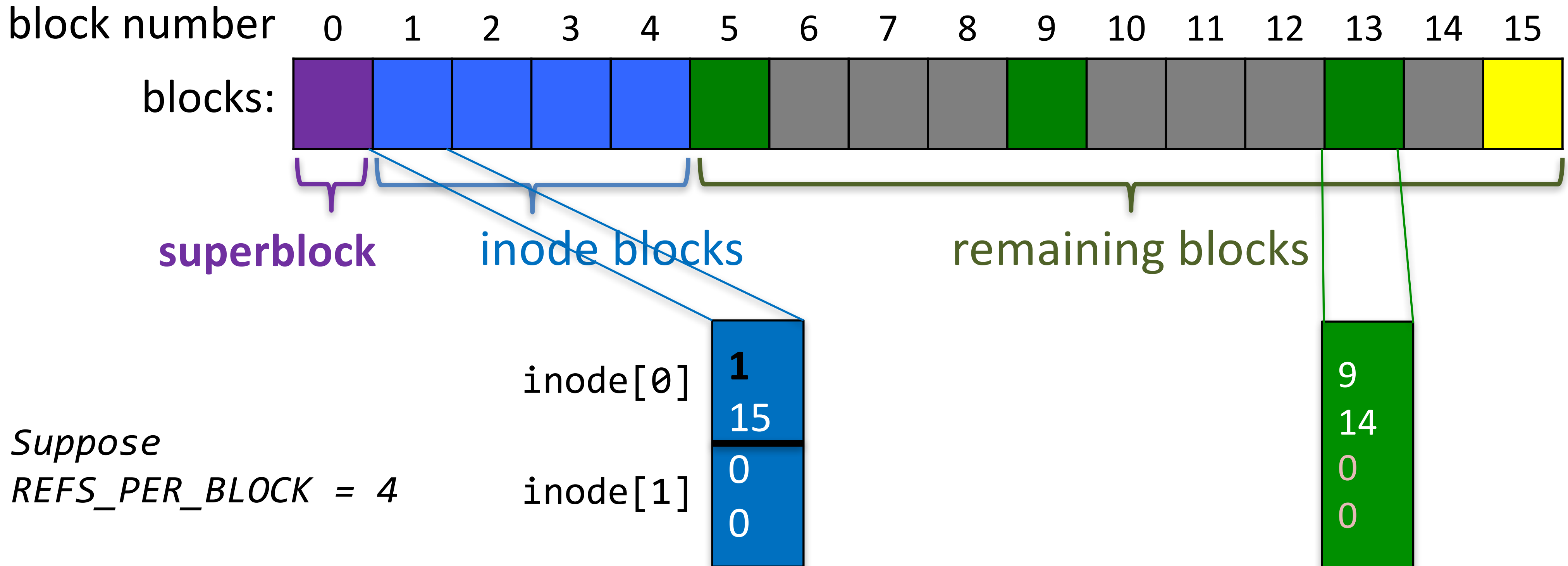
refs[0]: # of another freelistblock or **0 if end of list**

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

treedisk free list



treedisk I-node block



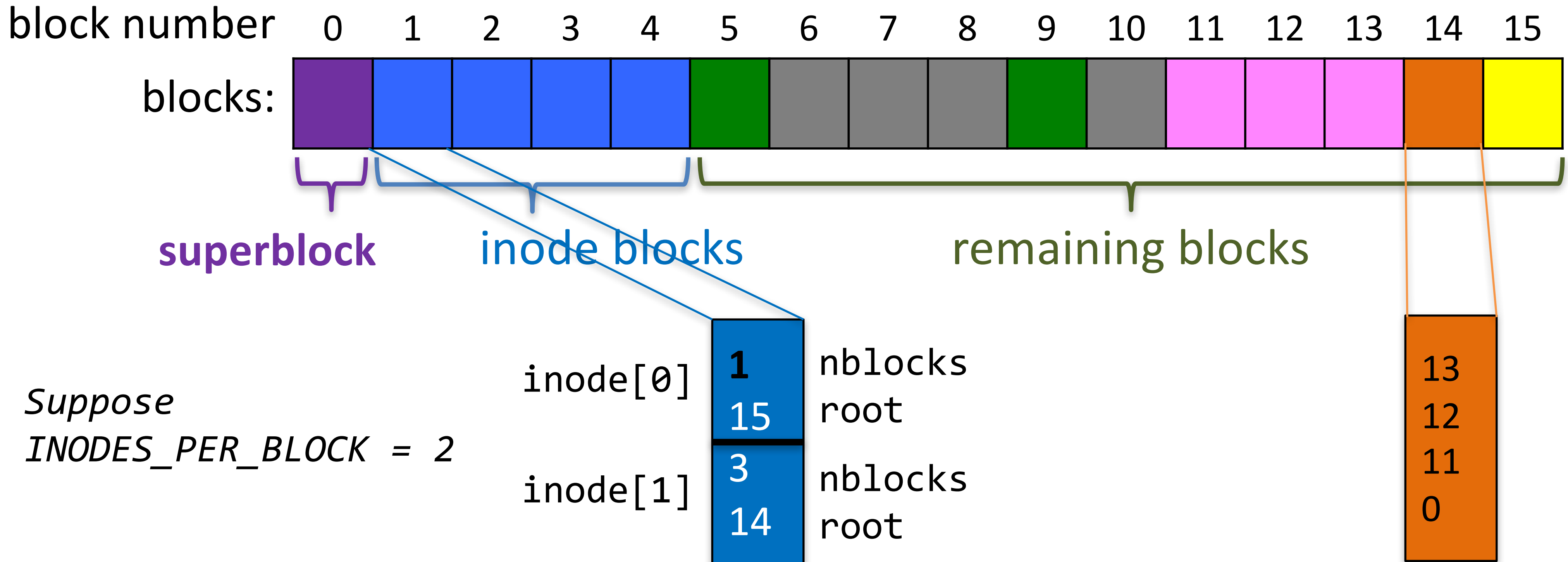
Suppose
REFS_PER_BLOCK = 4

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

What if the file is bigger than 1 block?

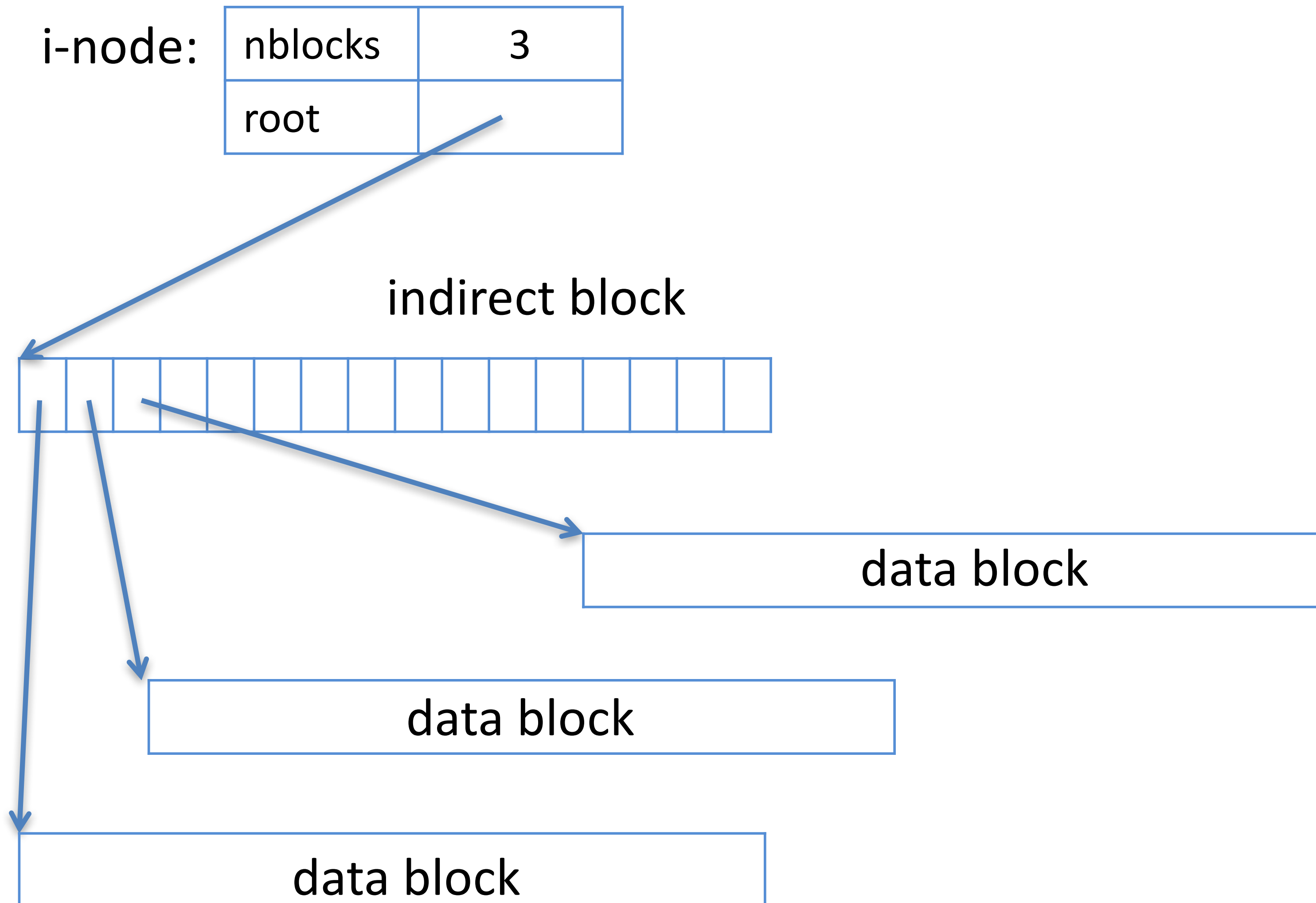
```
struct treedisk_inode {
  block_no nblocks;     // # blocks in virtual block store
  block_no root;        // block # of root node of tree (or 0)
};
```

treedisk Indirect block



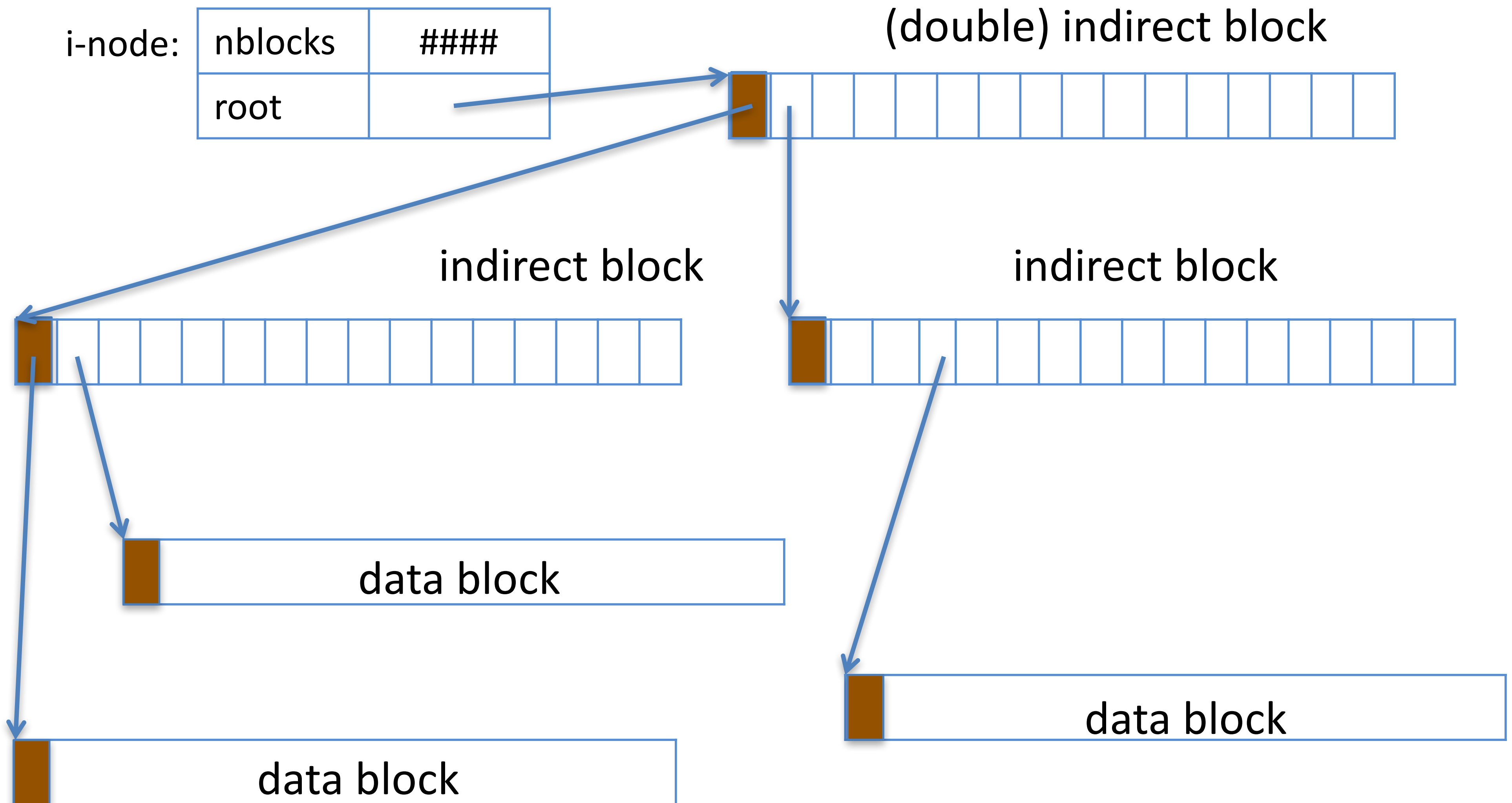

```
struct treedisk_indirblock {
    block_no refs[REFS_PER_BLOCK];
};
```

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

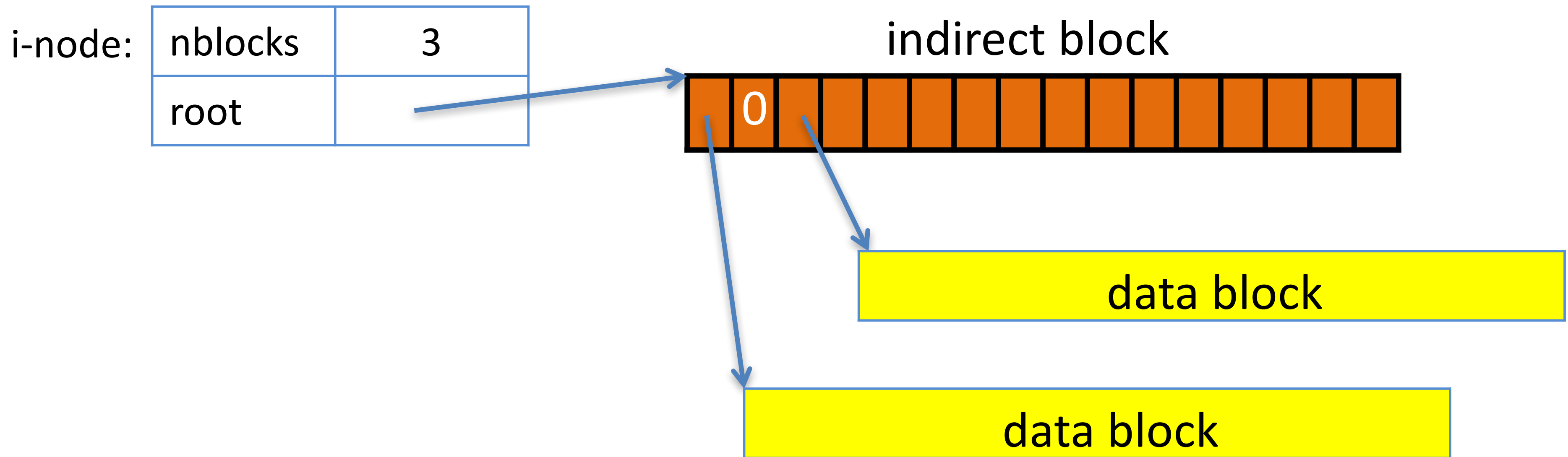
- all data blocks at bottom level
- #levels: ceil($\log_{RPB}$(#blocks)) + 1

  RPB = REFS_PER_BLOCK
- For example, if rpb = 16:

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

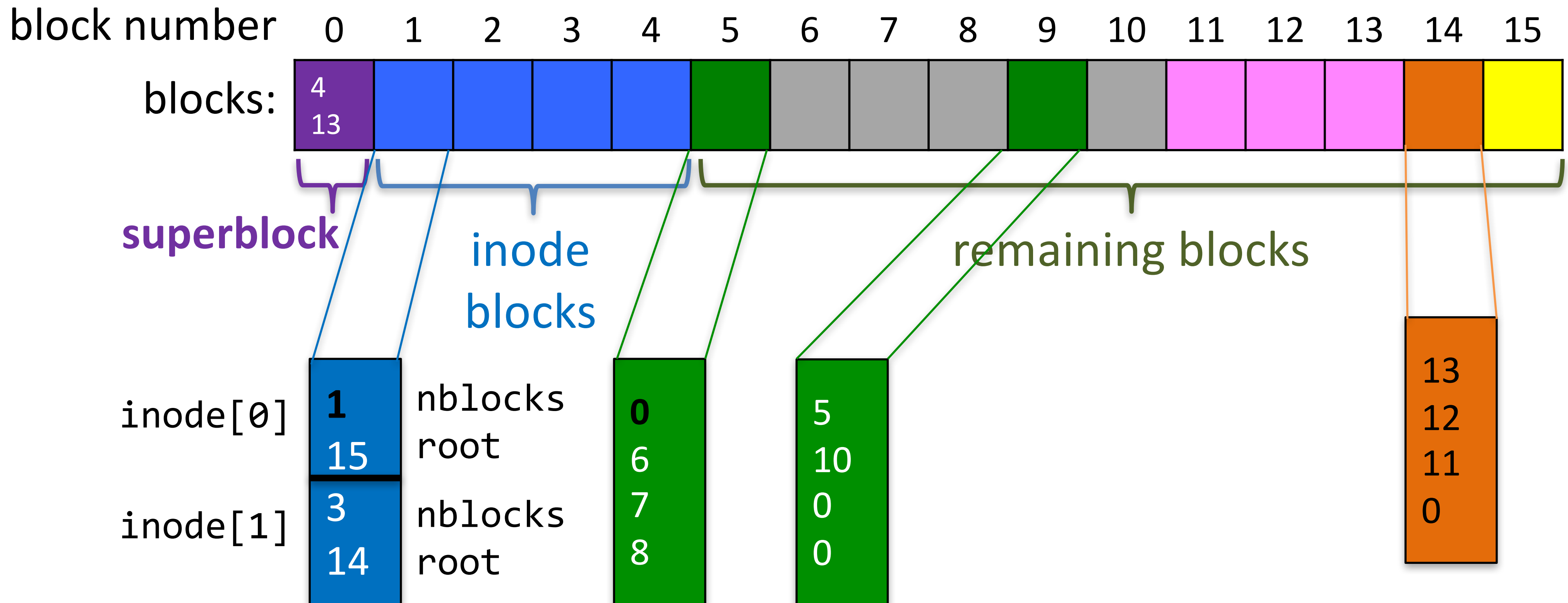
REFS_PER_BLOCK more commonly at least 128 or so

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

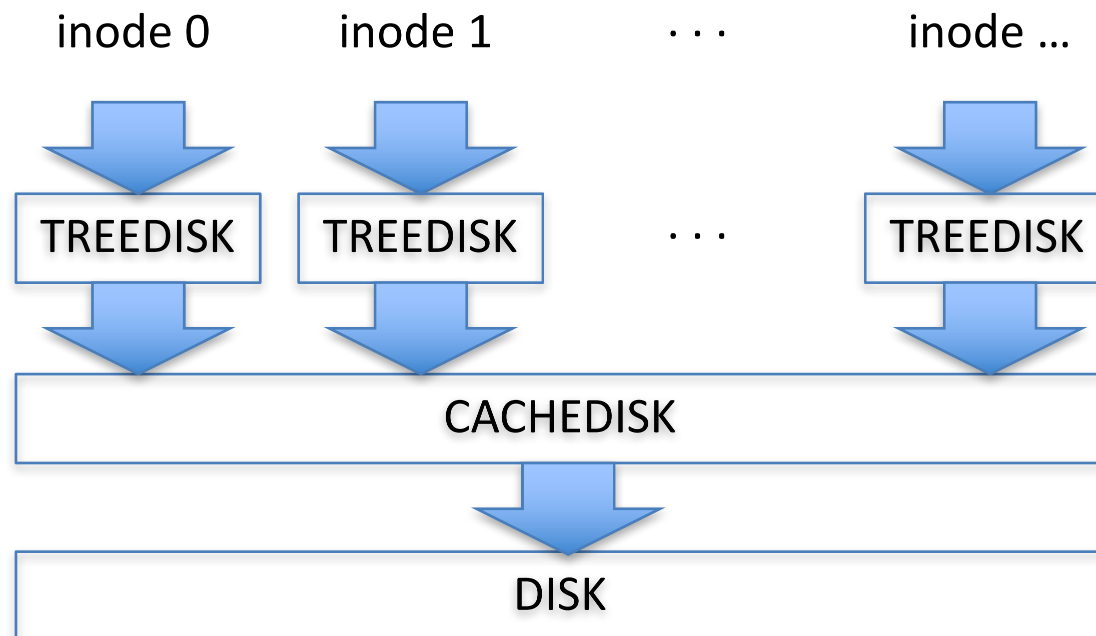
```
#define DISK_SIZE 1024
#define MAX_INODES 128

int main(){
  block_if 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

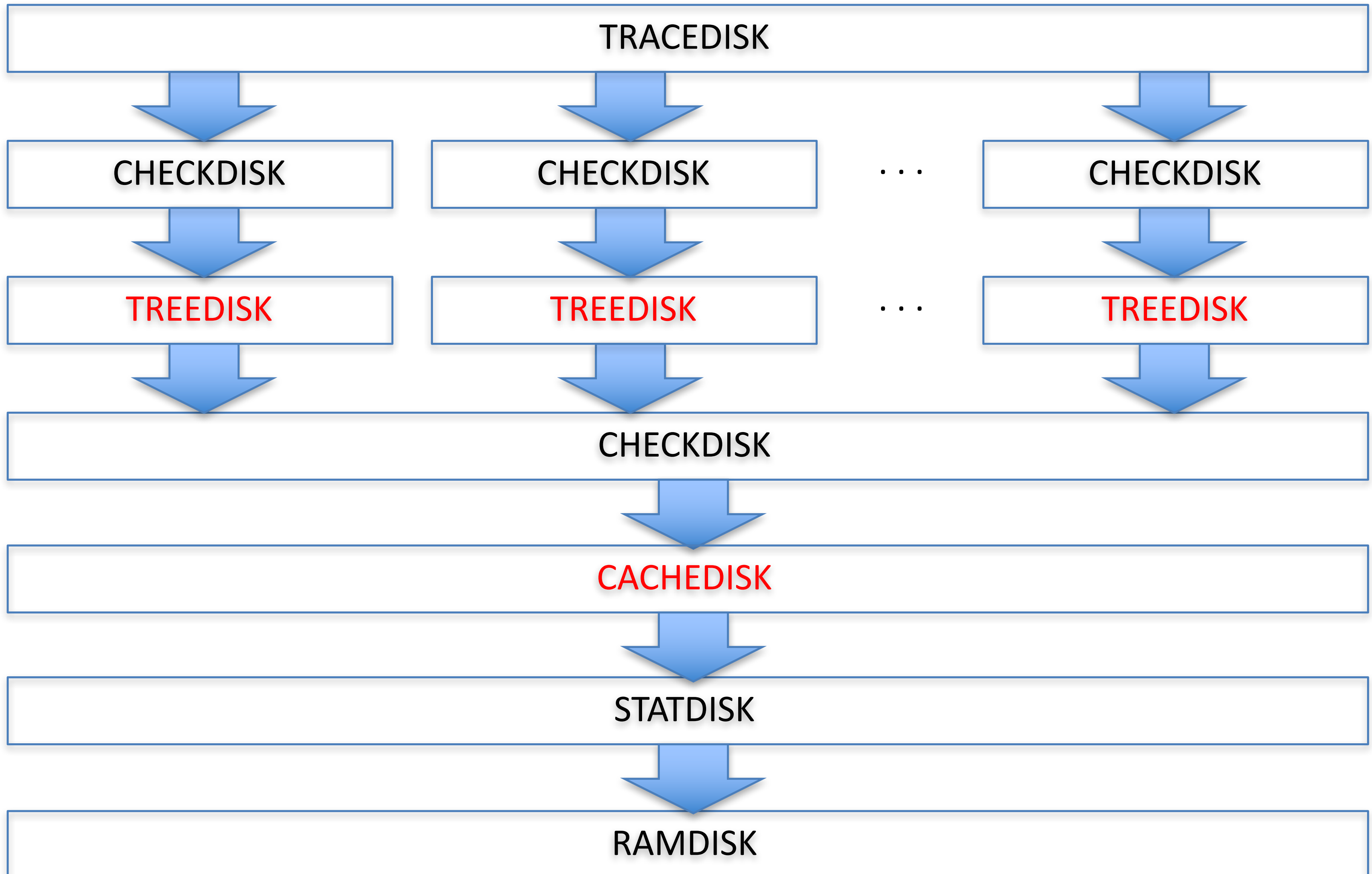
```
block_t cache[CACHE_SIZE];

int main(){
    block_if disk = disk_init(“disk.dev”, 1024);
    block_if cdisk = cachedisk_init(disk, cache, CACHE_SIZE);
    block_if disk0 = treedisk_init(cdisk, 0);
    block_if 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_if treedisk_init(block_if below,
                       unsigned int inode_no);
```



trace utility



tracedisk

- disk and ramdisk are 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_if tracedisk_init(
    block_if 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_if.o cachedisk.o checkdisk.o clockdisk.o
debugdisk.o disk.o ramdisk.o statdisk.o tracedisk.o treedisk.o
treedisk_chk.o
$ ./trace
blocksize:  512
refs/block: 128
!!TDERR: setsize not yet supported
!!ERROR: tracedisk_run: setsize(1, 0) failed
!!CHKSIZE 10: nblocks 1: 0 != 2
!$STAT: #nnblocks:  5 ← bug!
!$STAT: #nsetsize:  0
!$STAT: #nread:     32
!$STAT: #nwrite:    20
```

```
Trace
W:0:0
N:0:1
W:0:1
N:0:2
W:1:0
N:1:1
W:1:1
N:1:2
S:1:0
N:1:0
```

Cmd:inode:block

10-P4: 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

10-P4: 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!!*
- clockdisk.c is provided
  - it implements the CLOCK algorithm
  - you can implement a refined version of CLOCK, like a two-handed clock
  - consult the web for caching algorithms!

10-P4: Part 3/3

Implement your own trace file

- read, write, setsize operations
- at most 10,000 commands
- at most 128 inodes
- at most 1<<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 10-P4, but you may win fame and glory