MP4: Layered Block-Structured File System

Prof. van Renesse

Intro

- Underneath any file system, database system, etc. there are one or more *block stores*
- A block store provides a disk-like interface:
  - it is a sequence of blocks
    - typically a few kilobytes
  - you can read or write a block at a time
- The block store abstraction doesn't deal with file naming
- Last project: working with a block store abstraction

Block Store Abstraction

- Sequence of blocks numbered 0, 1, …
- Simple interface:
  - block_t block
    - block of size BLOCK_SIZE
  - 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

Simple block stores

- “disk”: a simulated disk stored on a Linux file
  - block_if bif = disk_init(char *filename, int nblocks)
    - Can 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
- block_if is a pointer to the block interface

Example code

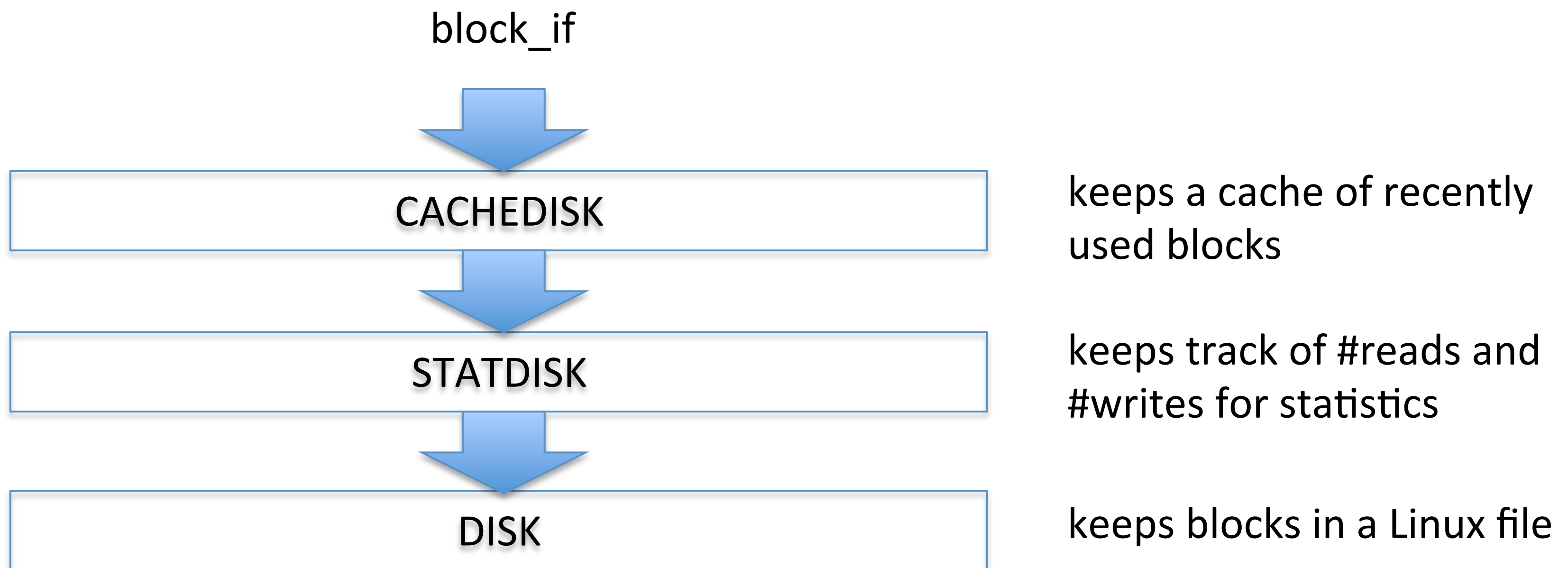
```
#include ...
#include “block_if.h”

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;
}
```

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);
};
```

Block Stores can be Layered!

Each layer presents a block_if 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;
}
```

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
```

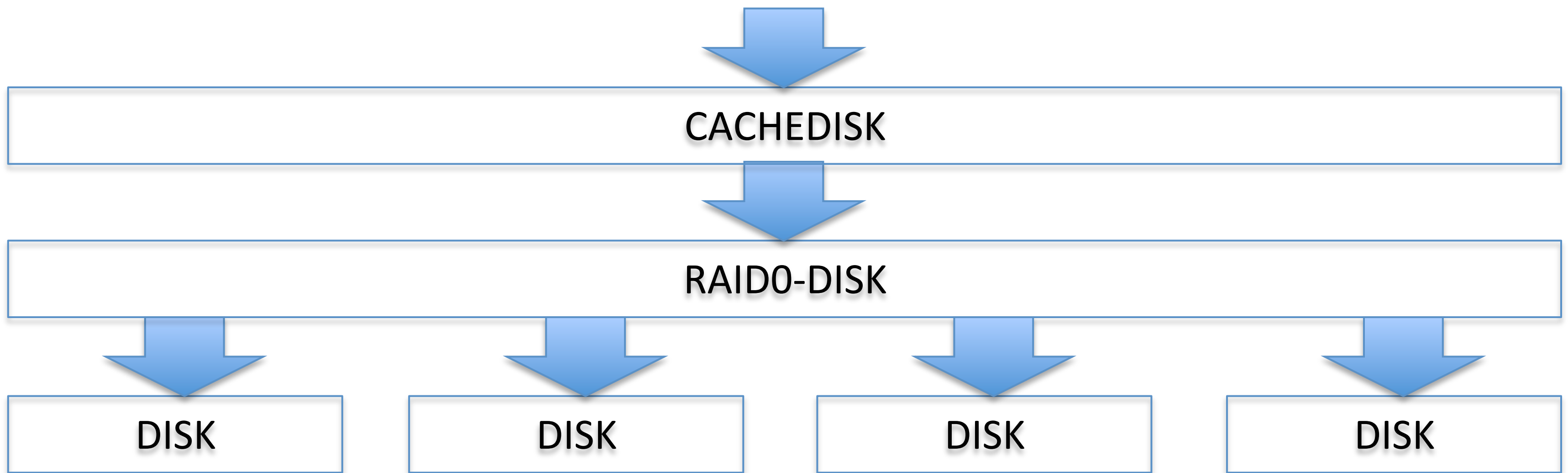
How to write a layer

```
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

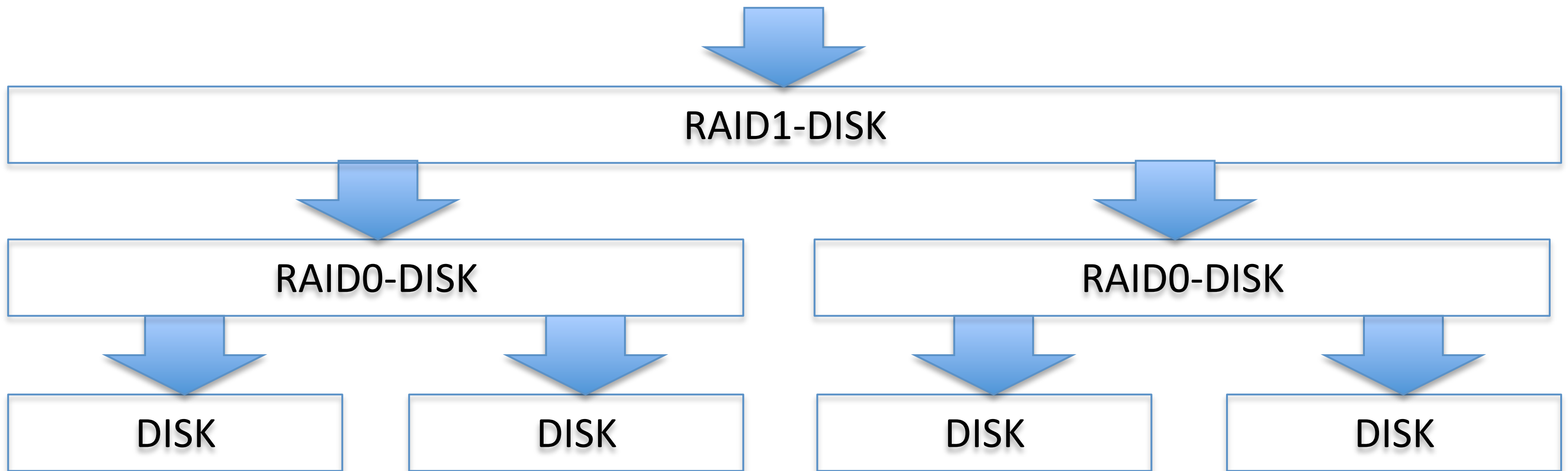
```
static int statdisk_read(block_if bi, block_no offset, block_t *block){
    struct statdisk_state *sds = bi->state;
    sds->nread++;
    return (*sds->below->read)(sds->below, offset, block);
}

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

static int statdisk_nblocks(block_if bi){ ... }
static int statdisk_setsize(block_if bi, block_no nblocks){ ... }

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

RAID 0



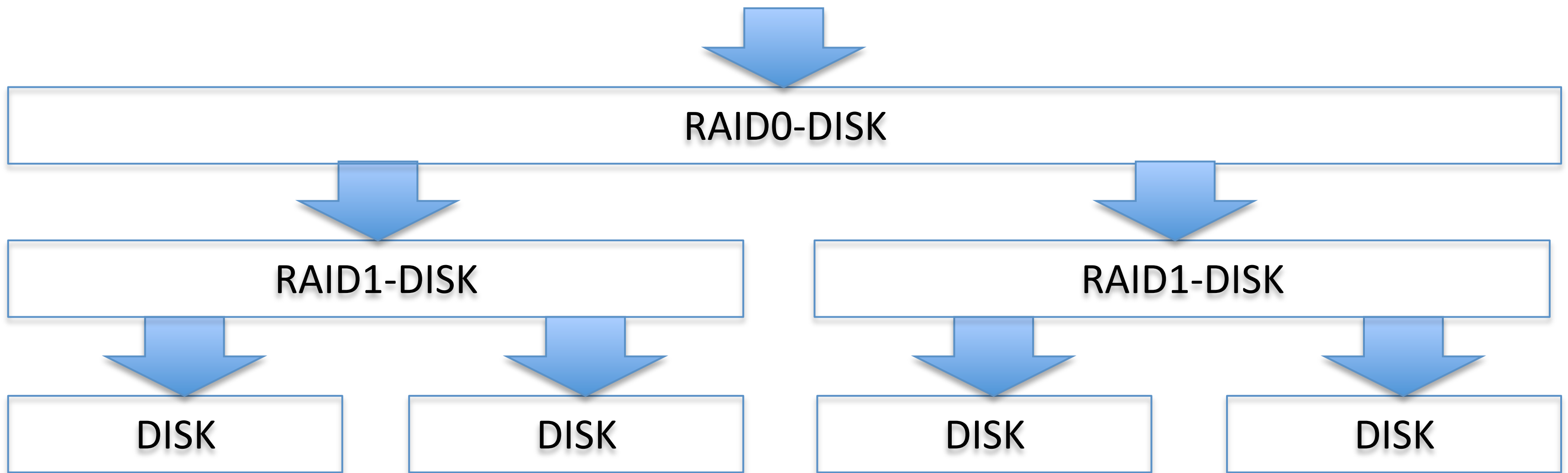

```
block_if raid0disk_init(block_if *below, unsigned int nbelow);
```

RAID 0+1




```
block_if raid1disk_init(block_if *below, unsigned int nbelow);
```

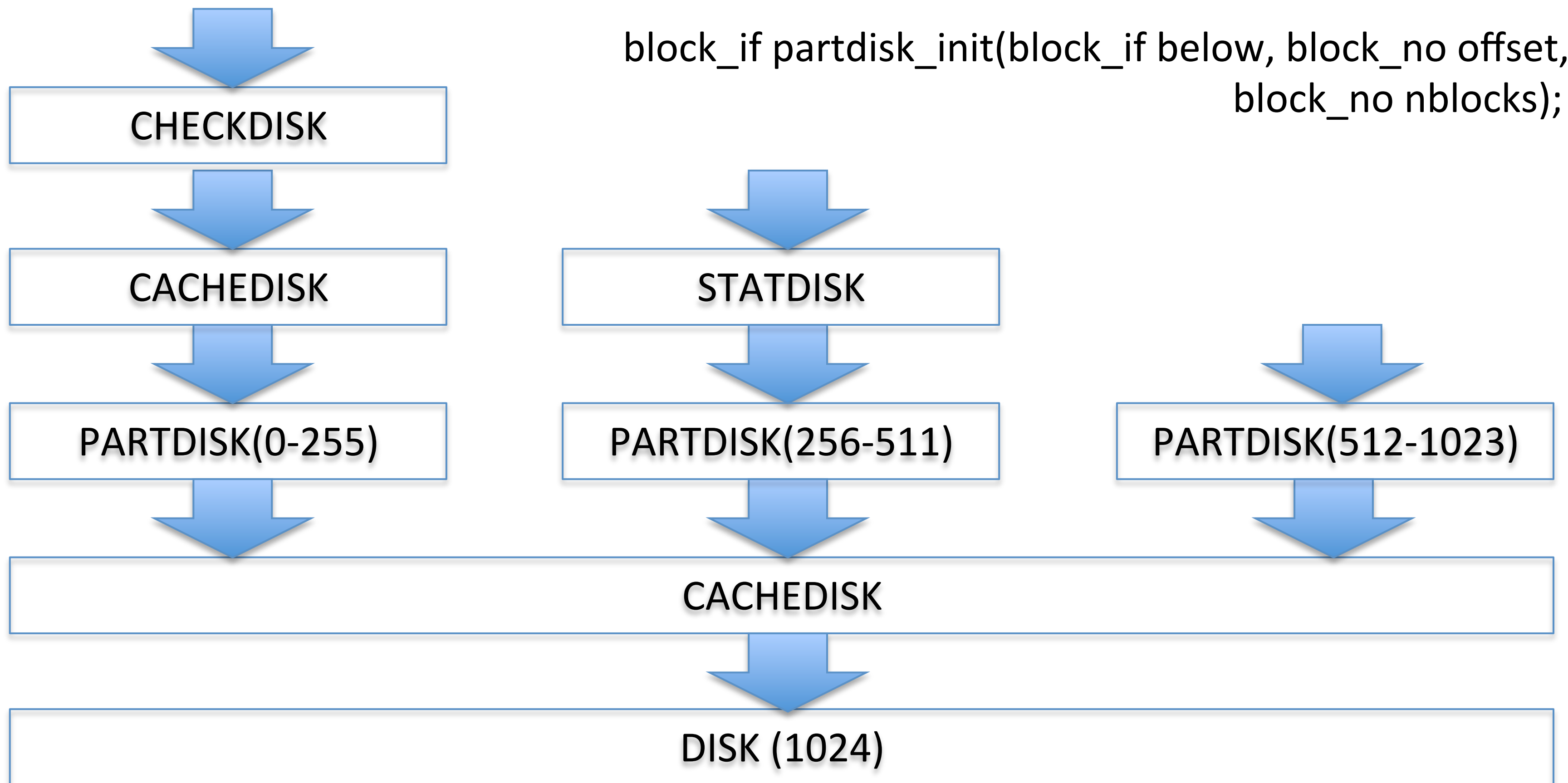
RAID 1+0



Multiplexing

- A single block store can be “multiplexed”, offering multiple virtual block stores
  - – opposite of RAID
- One way is simply partitioning the underlying block store into multiple disjoint sections
  - – partdisk

Partitioning: Cactus Stack



In general, layers form a Directed Acyclic Graph

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!

Partitioning with *treedisk*

- treedisk is a file system, somewhat similar to Unix file systems
- Offers multiple virtual block stores
- The underlying block store is partitioned into three sections:
  1. *superblock*
     - at block #0
  2. a fixed number of *i-node blocks*
     - start at block #1
     - the number is given in the superblock
  3. the remaining blocks
     - start at 1 + #i-node blocks
     - *data blocks, free blocks, indirect blocks, freelist blocks*

treedisk: layout

block number 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

blocks:

super block

inode blocks

remaining blocks

treedisk superblock

(one per underlying block store)

```
struct treedisk_superblock {
    block_no n_inodeblocks;
        // number of i-node blocks

    block_no free_list;
        // first block on the free list
};
```

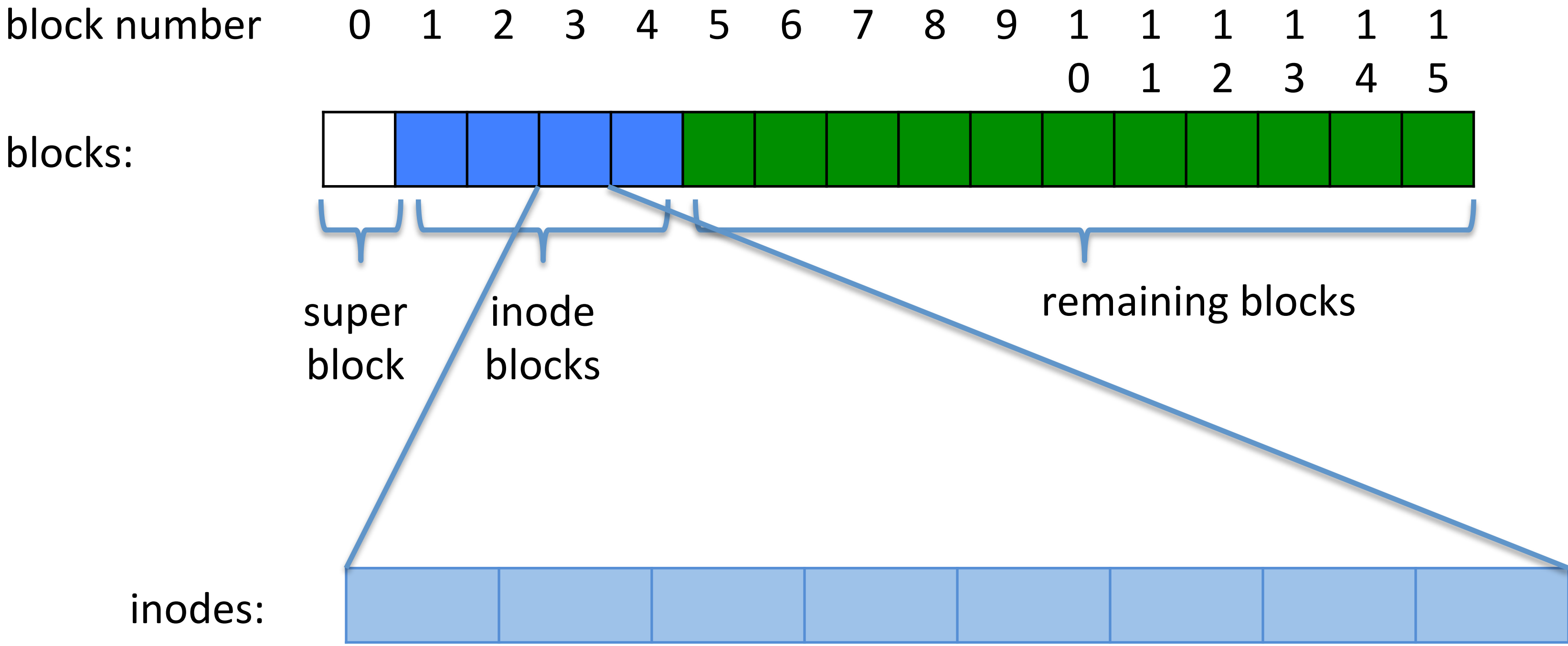
treedisk i-node
(one per virtual block store)

```
struct treedisk_inode {
    block_no nblocks;
            // #blocks in the virtual block store

    block_no root;
            // block number of root node of tree
            // should be 0 if nblocks == 0
};
```

treedisk i-node block

```
#define INODES_PER_BLOCK    (BLOCK_SIZE /   \
              sizeof(struct treedisk_inode))

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

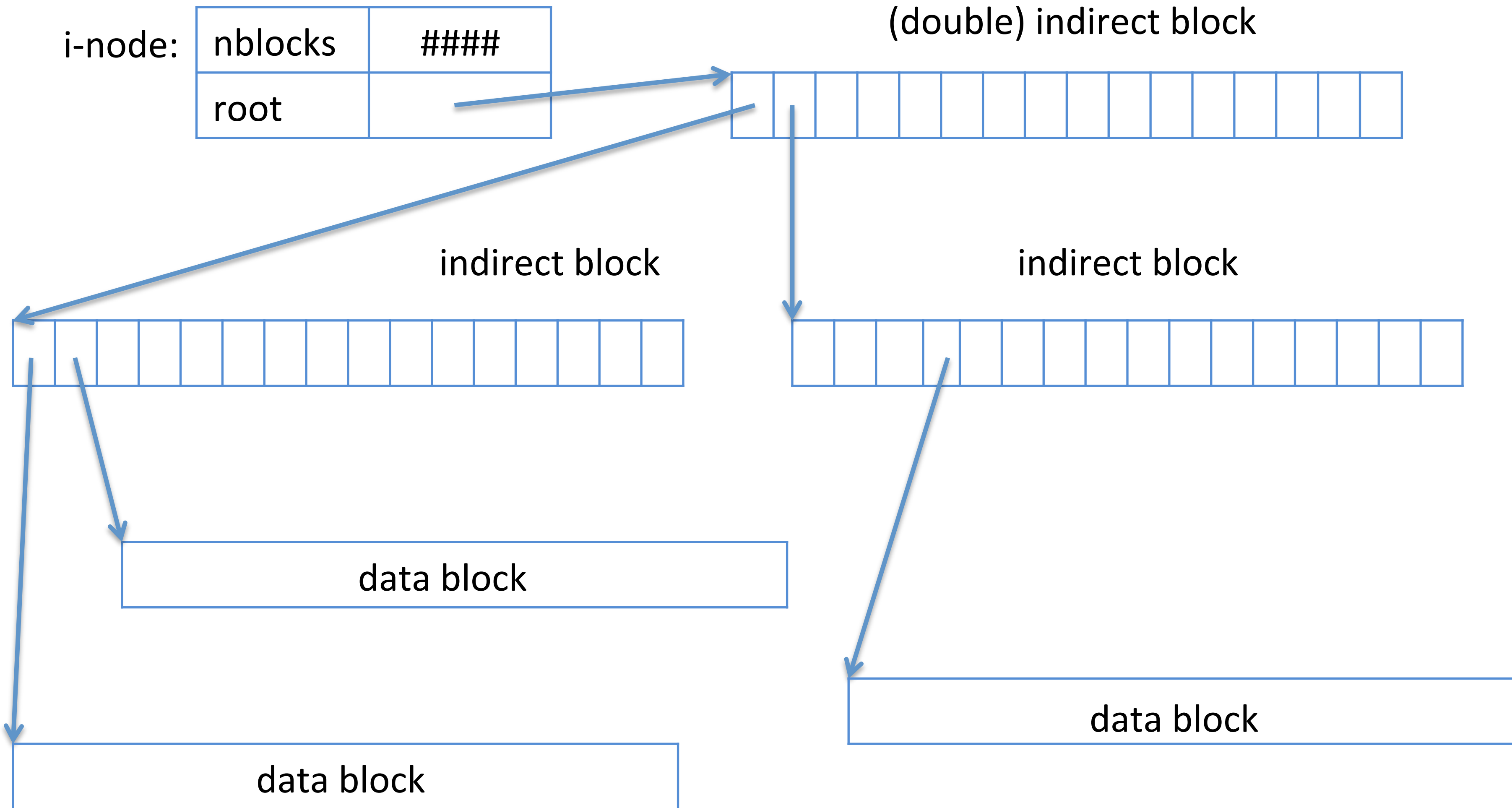
treedisk: layout



treedisk indirect block

```
#define REFS_PER_BLOCK        (BLOCK_SIZE /   \
                                sizeof(block_no))

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

treedisk virtual block store



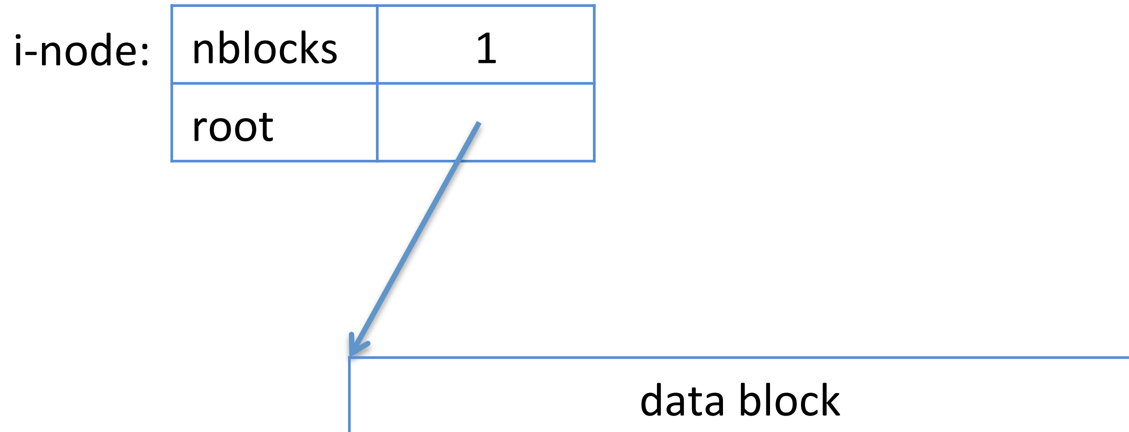
treedisk virtual block store

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

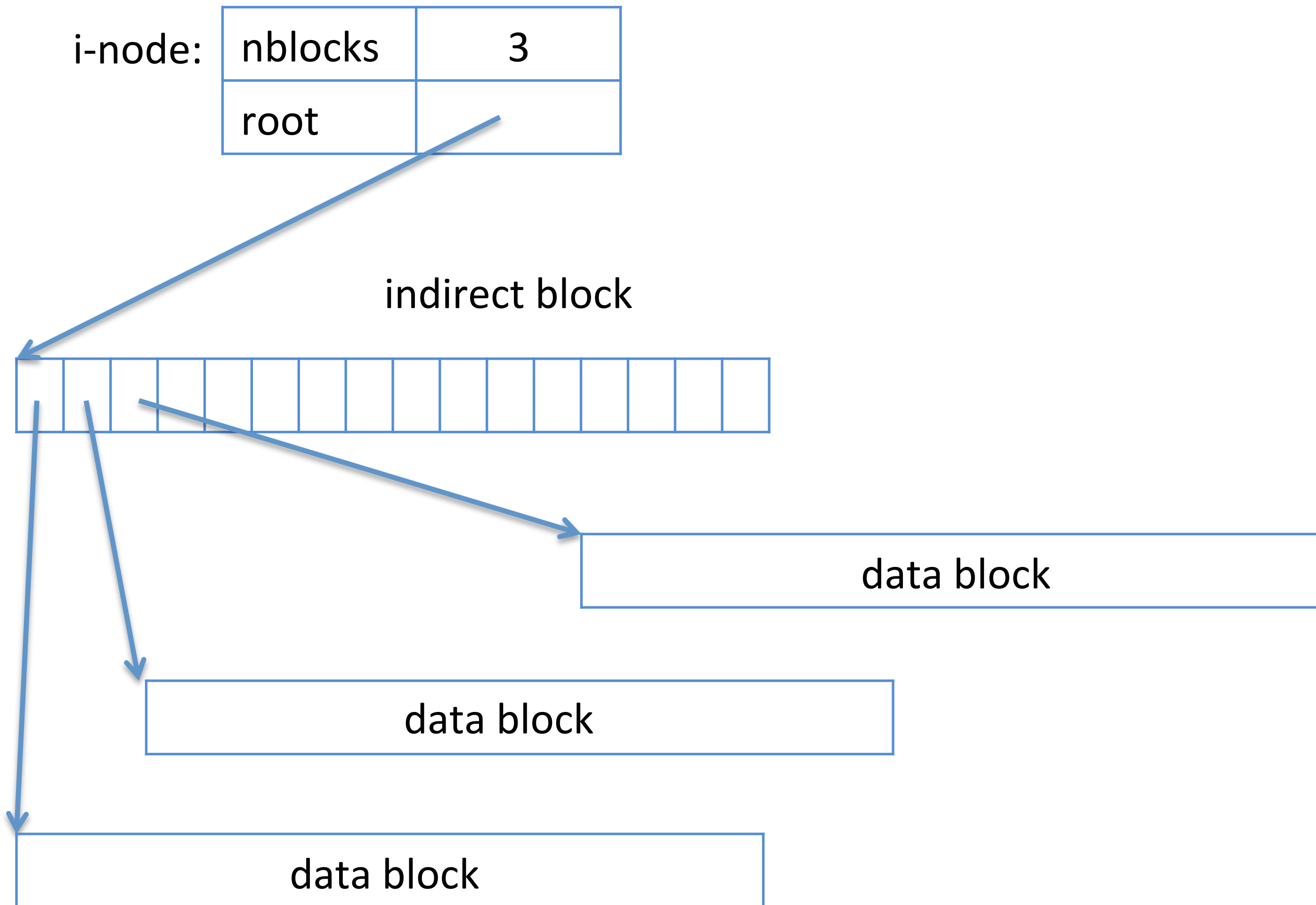
#blocks	#levels
0	0
1	1
2 - 16	2
17 - 256	3
257 - 4096	4

REFS_PER_BLOCK more commonly at least 128 or so

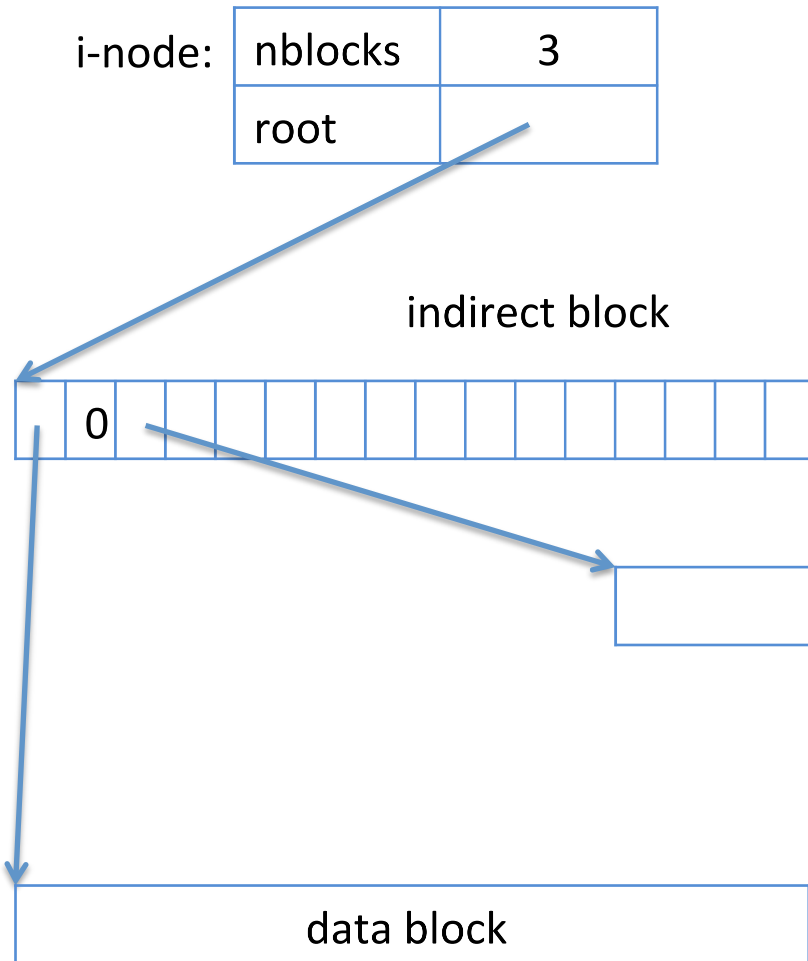
virtual block store: 1 block



virtual block store: 3 blocks



virtual block store: with hole



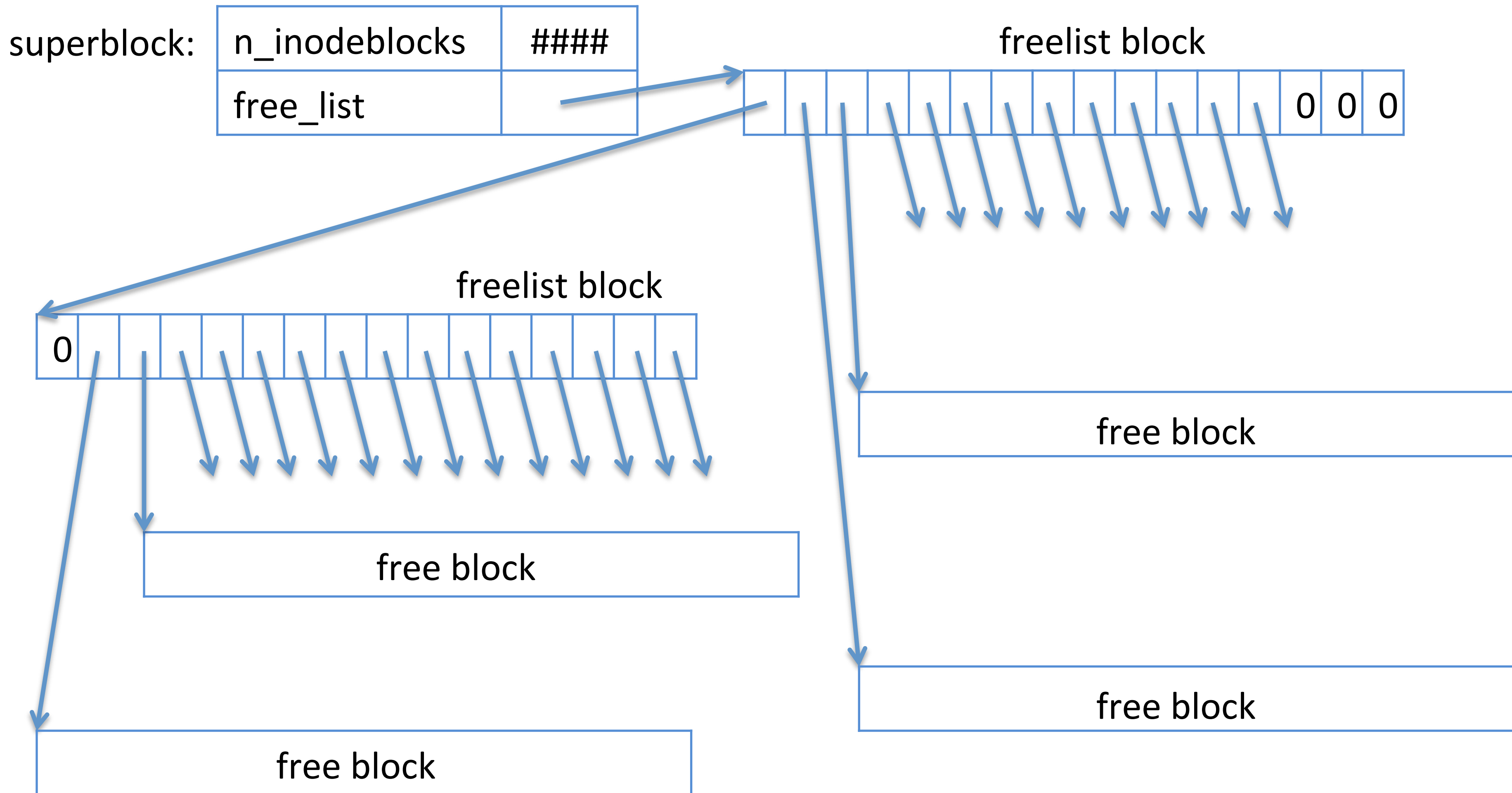
- Hole appears as a virtual block filled with null bytes
- pointer to indirect block can be 0 too
- virtual block store can be much larger than the “physical” block store underneath!

treedisk: free list

```
struct treedisk_superblock {
    block_no n_inodeblocks;
    block_no free_list;   // linked list
};

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

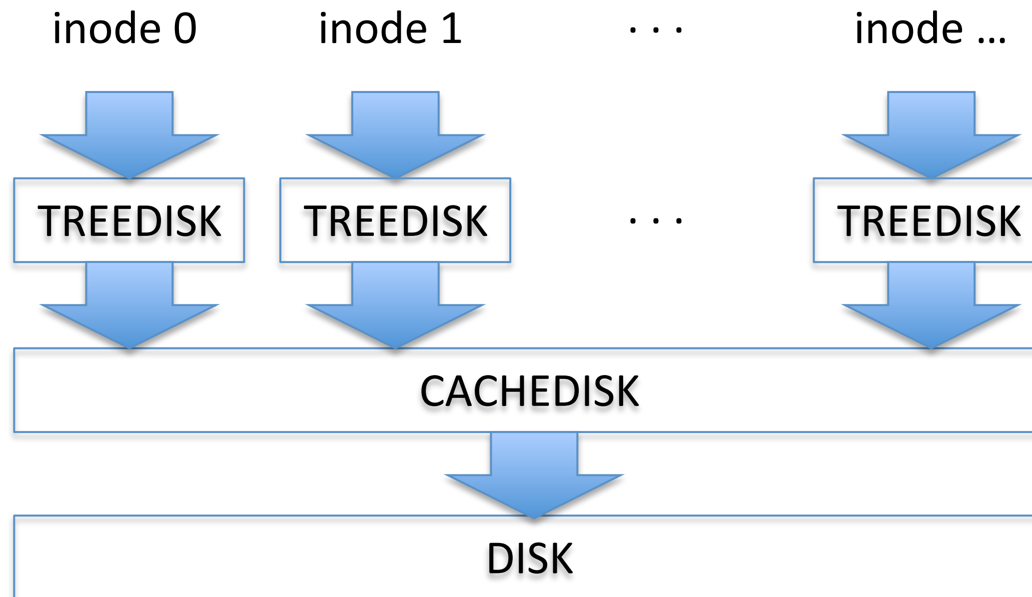
- refs[0] points to another freelistblock or to 0 if end of list
- refs[i] points to free block for all i > 1, or 0 if slot empty

treedisk free list

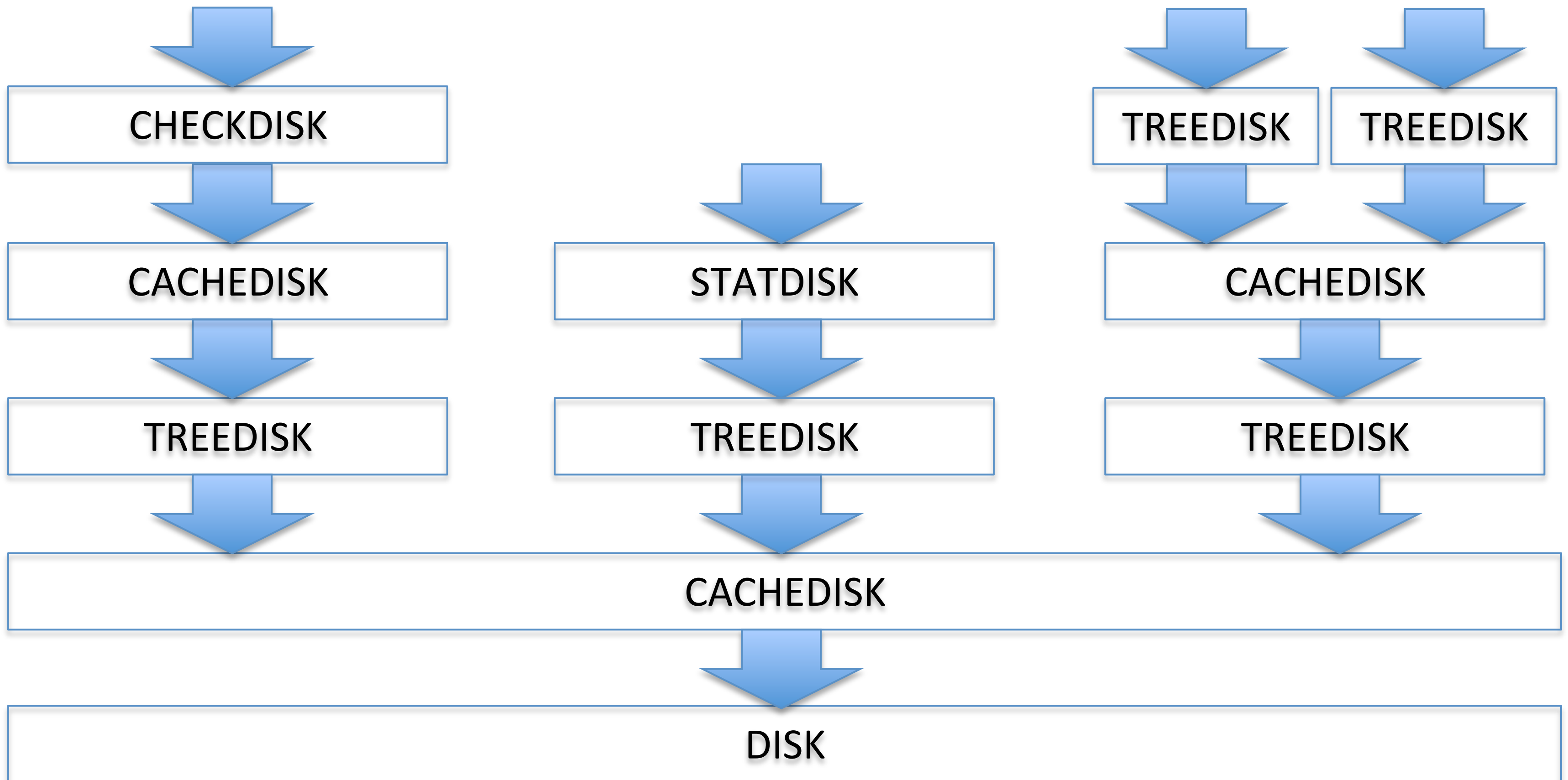


Layering on top of treedisk

```
block_if treedisk_init(block_if below,
                        unsigned int inode_no);
```



Cactus Stack



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;
}
```

But first: create a 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;
}
```

tracedisk

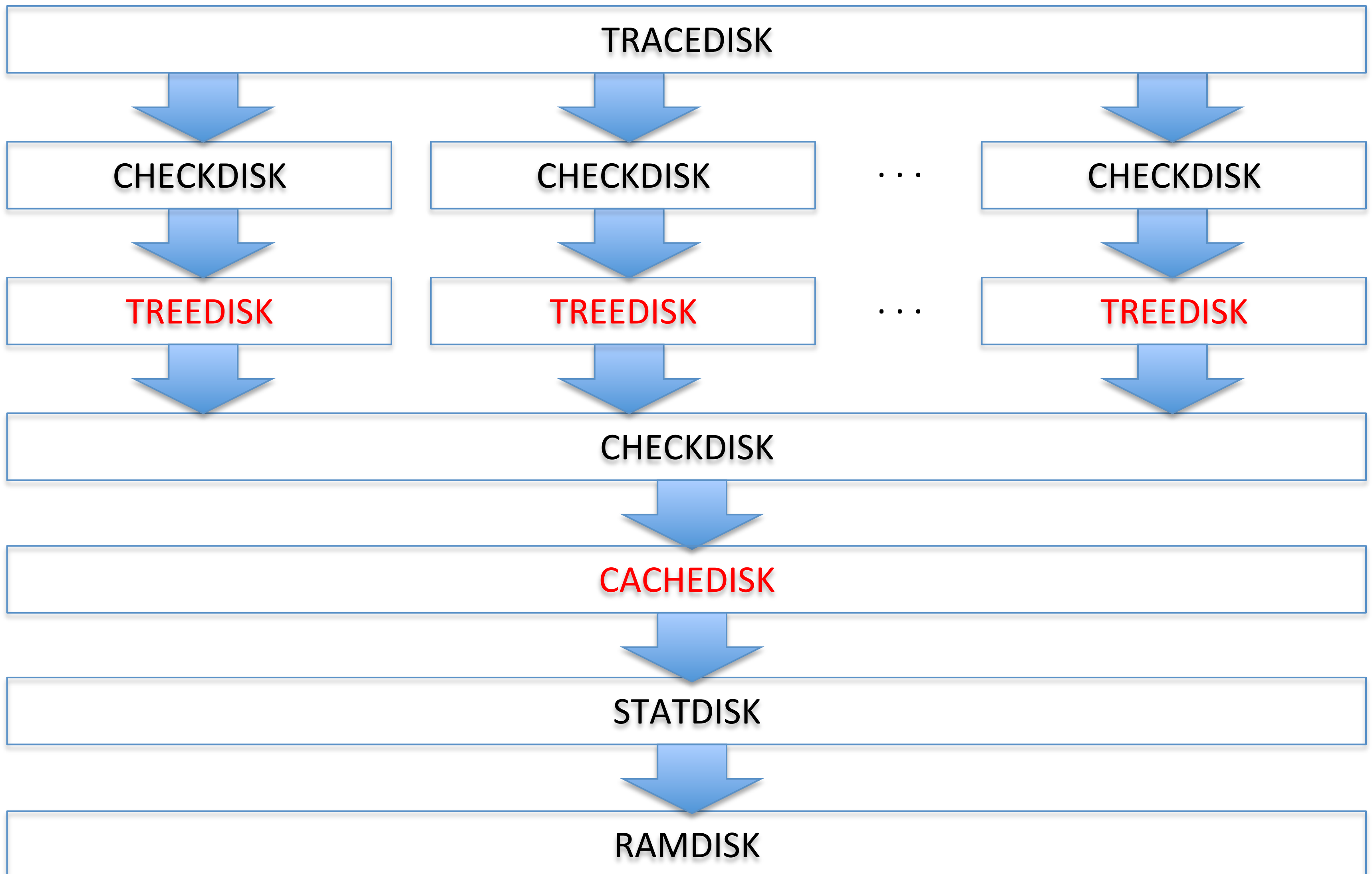
- disk and ramdisk are bottom-level block store
- tracedisk is a top-level block store
  - or "application-level" if you will
  - you can't layer on top of it

```
block_if tracedisk_init(
    block_if below,
    char *trace, // 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
```

trace utility



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 partdisk.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
!$STAT: #nnblocks:  0
!$STAT: #nsetsize:  0
!$STAT: #nread:     32
!$STAT: #nwrite:    20$
```

Homework: 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

Homework: 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!

Homework: 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 MP4, but you may win flexpoints and glory

Ideas for new layers

- Other block store file systems
  - Unix FS, LFS, FAT, your own design, …
  - (could be compatible with real file systems!)
- Compression, Deduplication
- Encryption, Parity, ECC (Hamming), …
- Fault Injection Disk
  - For testing: purposely flips bits, etc.
- Shared Network Disk
  - two layers
    - Bottom layer: clientdisk
    - Top layer: serverdisk
- Other RAID levels
- …

Other ideas

- Implement a POSIX-compatible interface to block stores
  - Implement directories, byte-addressable files, mounting, etc.