Intro to A4: Block Stores

CS 4410

Operating Systems

[A. Bracy, R. Van Renesse]

Introduction

File System

abstraction that provides persistent, *named* data

Block Store
(or **Block Cache**)

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)

A4 Concepts

- **Block Store Abstraction**
- Cache Disk
- Tree Disk

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

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

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

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;
```

← *poor man's class*

← *function pointers, AKA class methods*

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.c -- just a lone 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;
}
```


RUN IT! IT’S COOL!

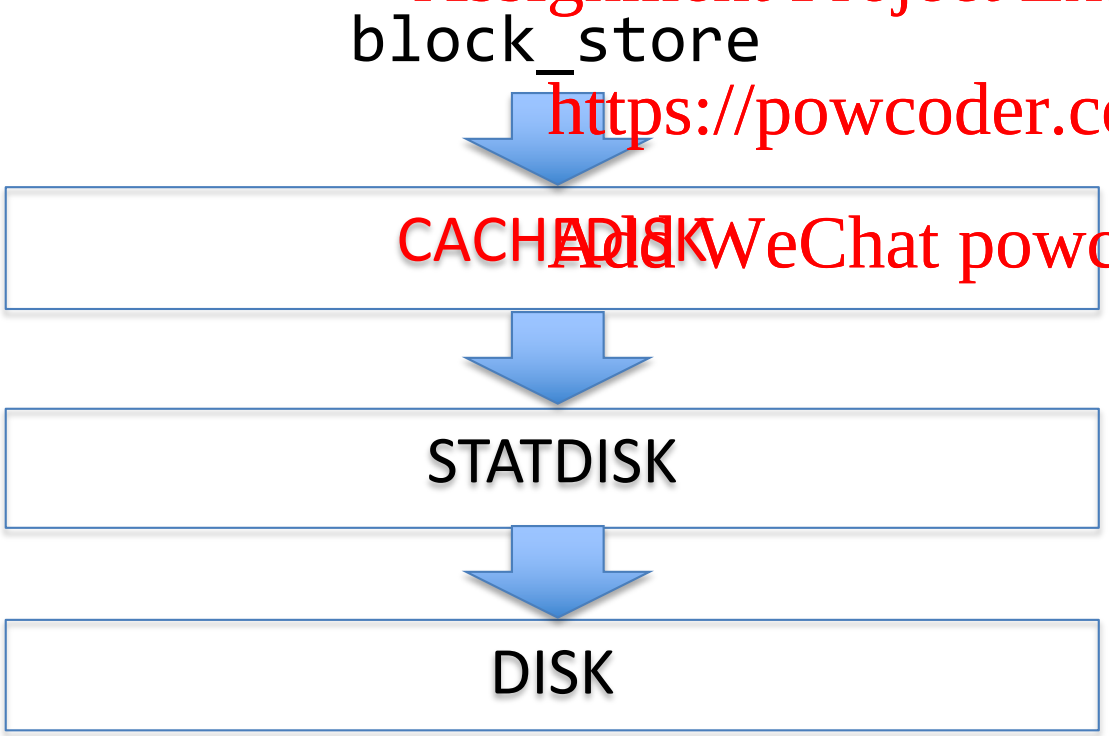
```
> gcc -g block_store.c sample.c
> ./a.out
> less disk.dev
```

A4 Concepts

- Block Store Abstraction
- **Cache Disk**
- Tree Disk

Block Stores can be Layered!

Each layer presents a block store abstraction



keeps a cache of recently used blocks

keeps track of #reads and #writes for statistics

keeps blocks in a Linux file

A Cache for the Disk? Yes!

All requests for a given block go through block cache

File System
AKA treedisk

Block Cache
AKA cachedisk

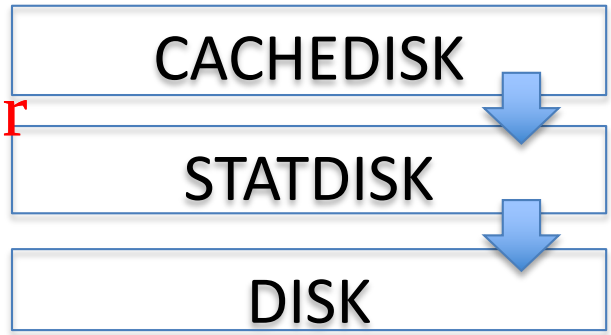
Disk

- 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

layer.c -- code with layers

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

block_t cache[CACHE_SIZE];

int main(){
     block_store_t *disk = disk_init("disk2.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, "Farewell World!");
     (*cdisk->write)(cdisk, 0, &block);
     (*cdisk->destroy)(cdisk);
     (*sdisk->destroy)(sdisk);
     (*disk->destroy)(disk);

     return 0;
}
```

RUN IT! IT'S COOL!

```
> gcc -g block_store.c statdisk.c cachedisk.c layer.c
> ./a.out
> less disk2.dev
```

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
};
```

layer-specific data

```
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;
}
```

function pointers, AKA class methods

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);
}
```


records the stats and passes the request to the layer below

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

A4 Concepts

- Block Store Abstraction
- Cache Disk
- **Tree Disk**

Another Possible Layer: Treedisk

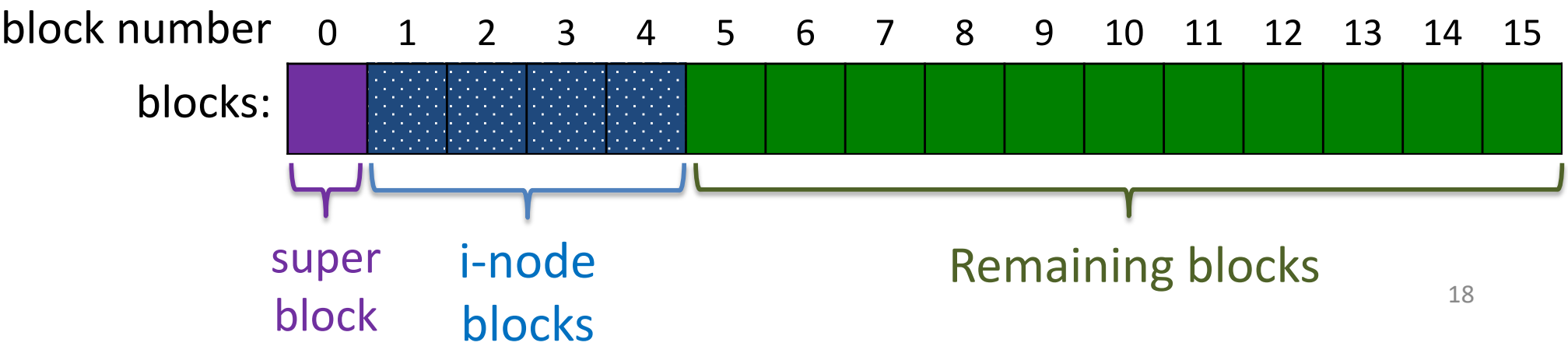
- A **file system**, similar to Unix file systems
- Initialized to support N virtual block stores (AKA files)
- Files are not named, but *numbered*


```
W:0:0        // write file 0, block 0
W:1:4        // write file 1, block 4
R:1:1        // read file 1, block 4
```

Treedisk Structure

Underlying block store (below) partitioned into 3 sections:

1. Superblock: block #0
2. Fixed number of *i-node blocks:* starts at block #1
3. Remaining blocks: starts after i-node blocks
   - *data blocks*
   - *indirect blocks*
   - *free blocks*
   - *freelist blocks*

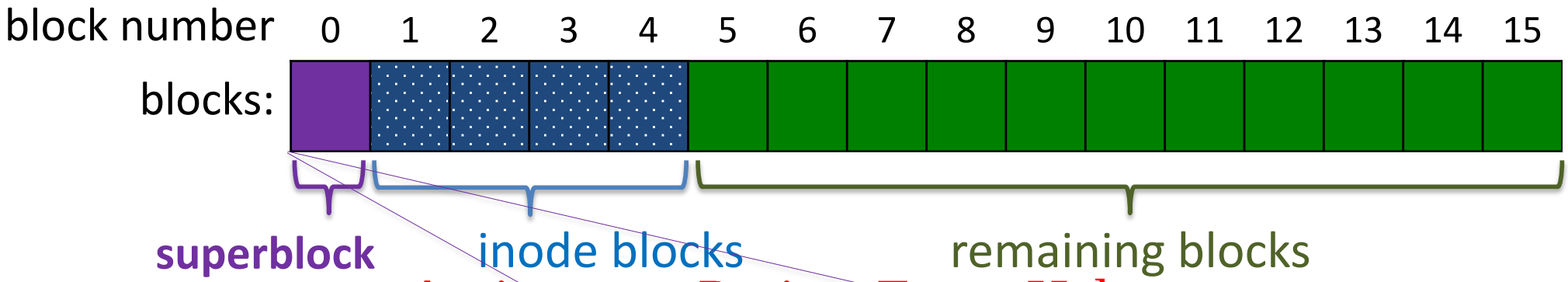


Types of Blocks in Treedisk

```
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
- I-nodeblock: list of inodes
- Indirblock: list of blocks
- Datablock: just data
- Freelistblock: list of all unused blocks below

treedisk Superblock

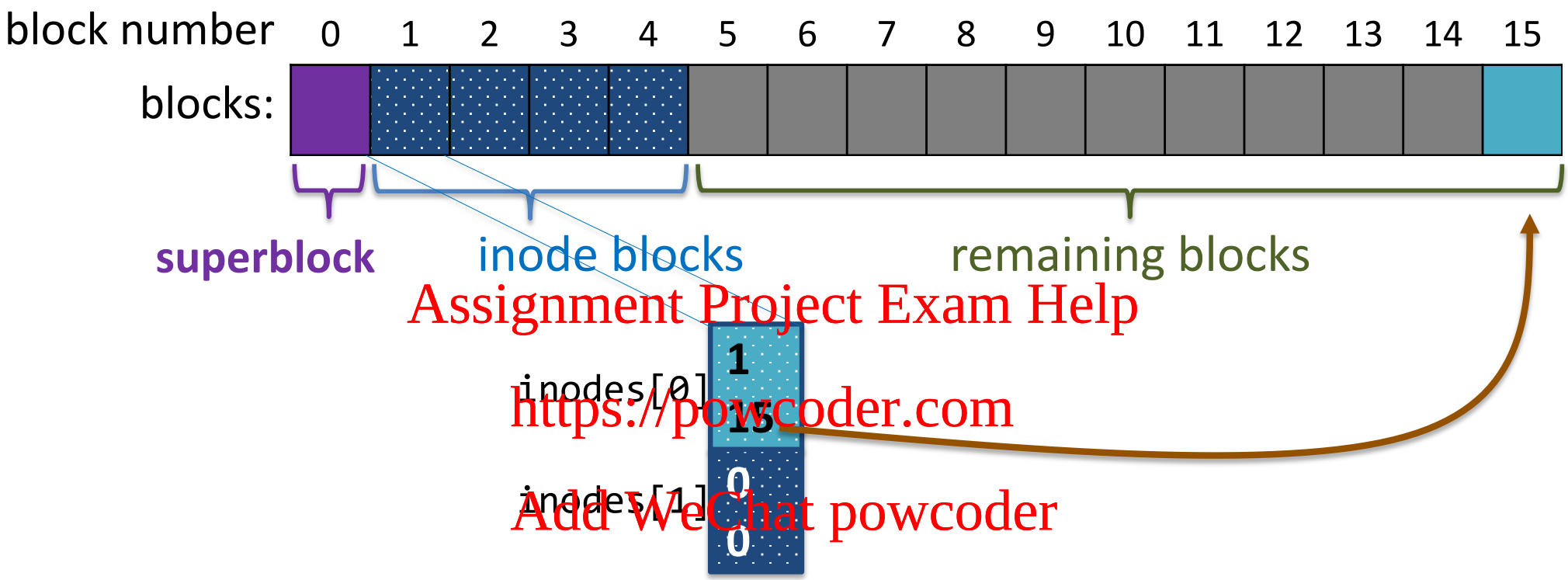



```
n_inodeblocks  4
free_list      ?
(some green box)
```

```
// 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 i-node block

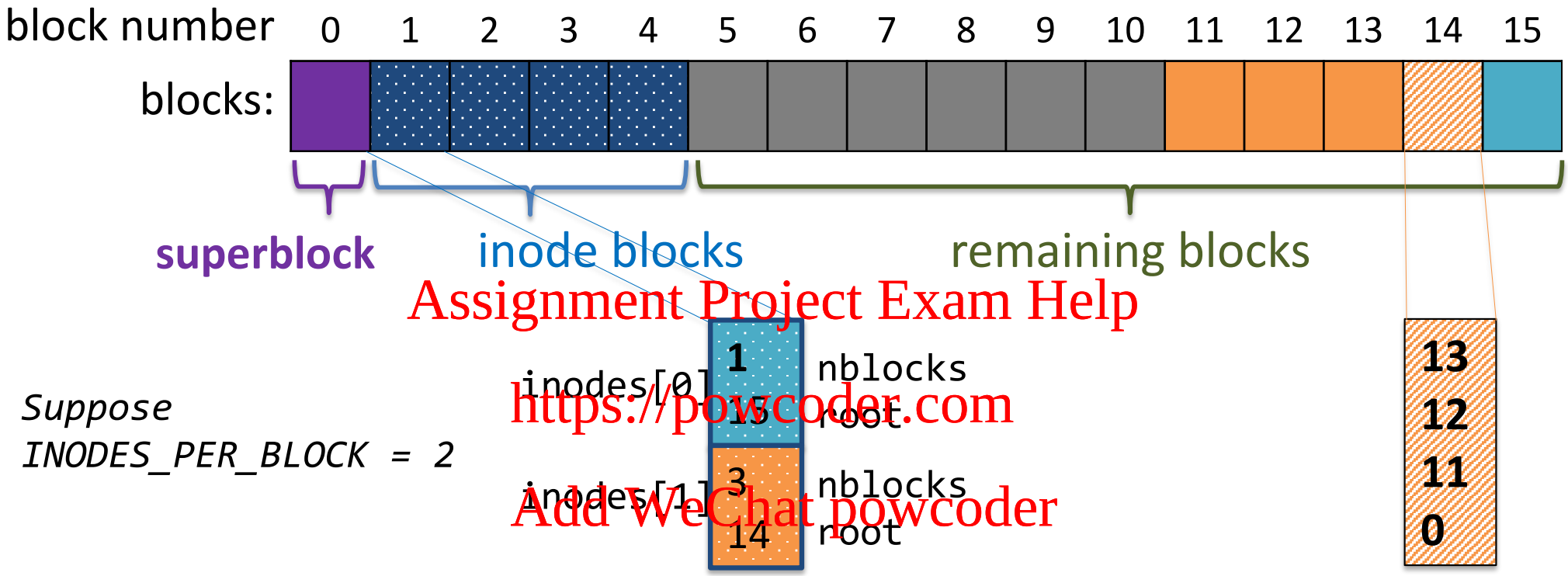



```
struct treedisk_inodeblock {
    struct treedisk_inode inodes[INODES_PER_BLOCK];
};
```

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)
};
```

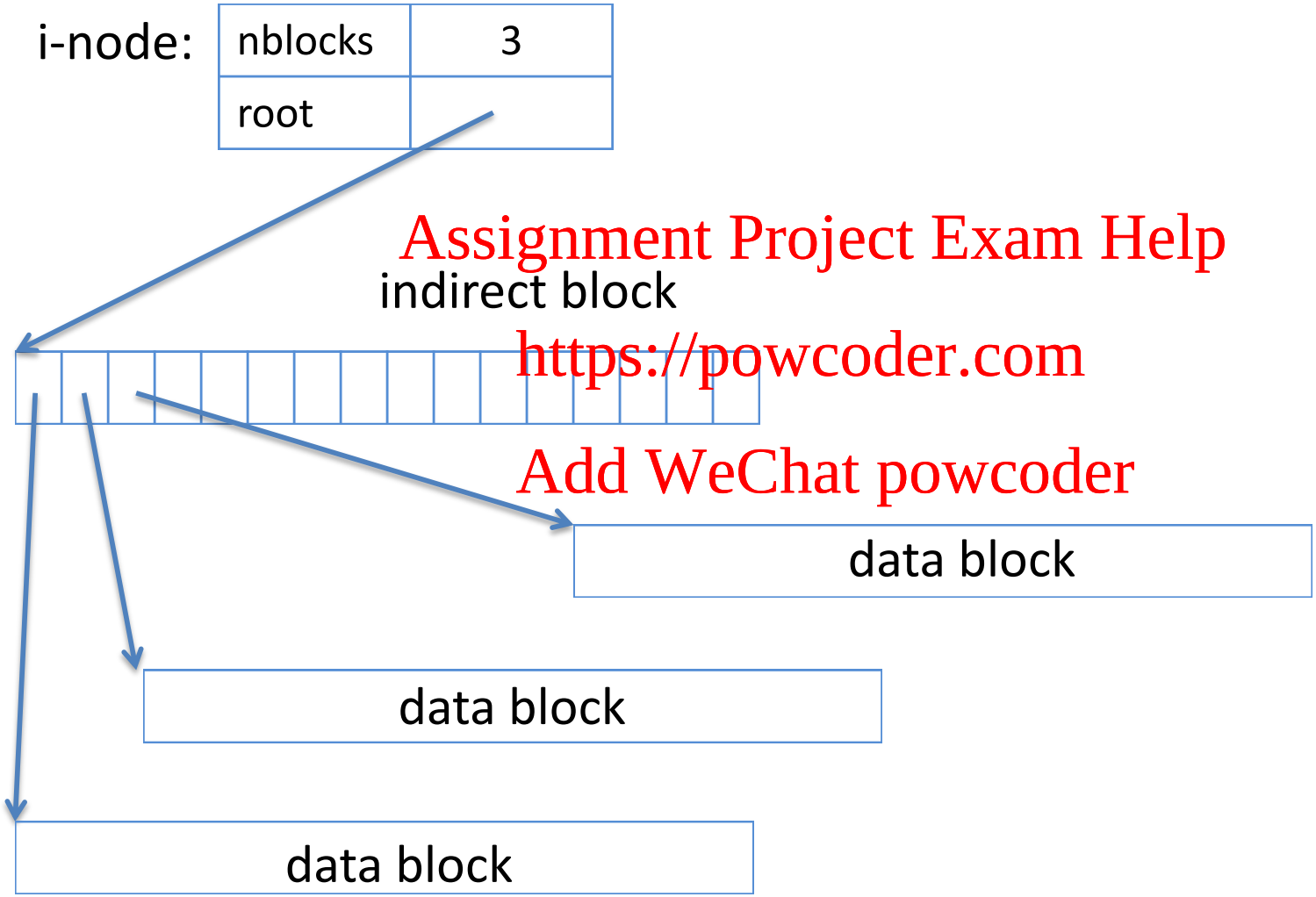
treedisk Indirect block



Suppose
INODES_PER_BLOCK = 2

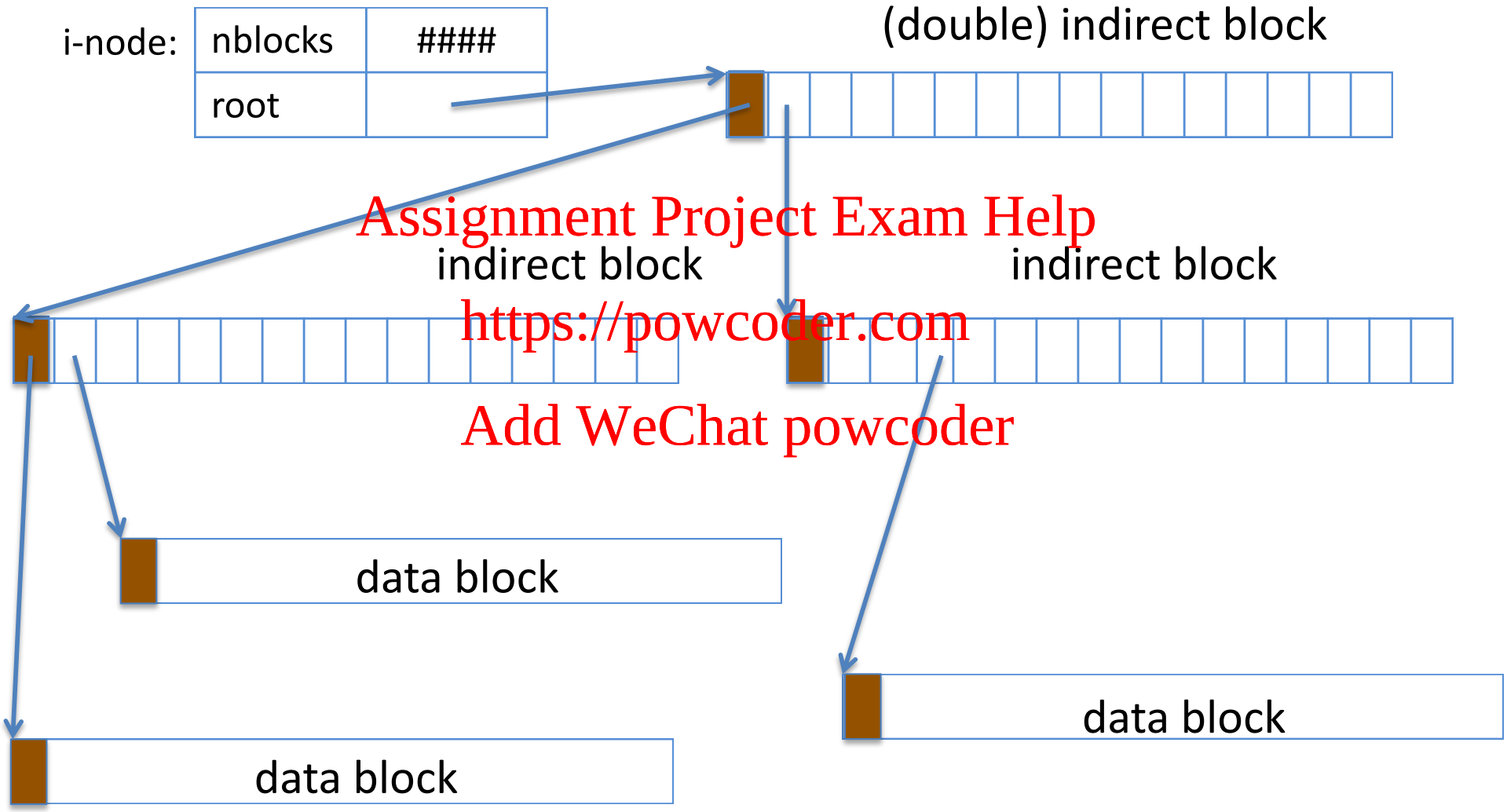
```
struct treedisk_indirblock {
    block_no refs[REFS_PER_BLOCK];
};
```

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

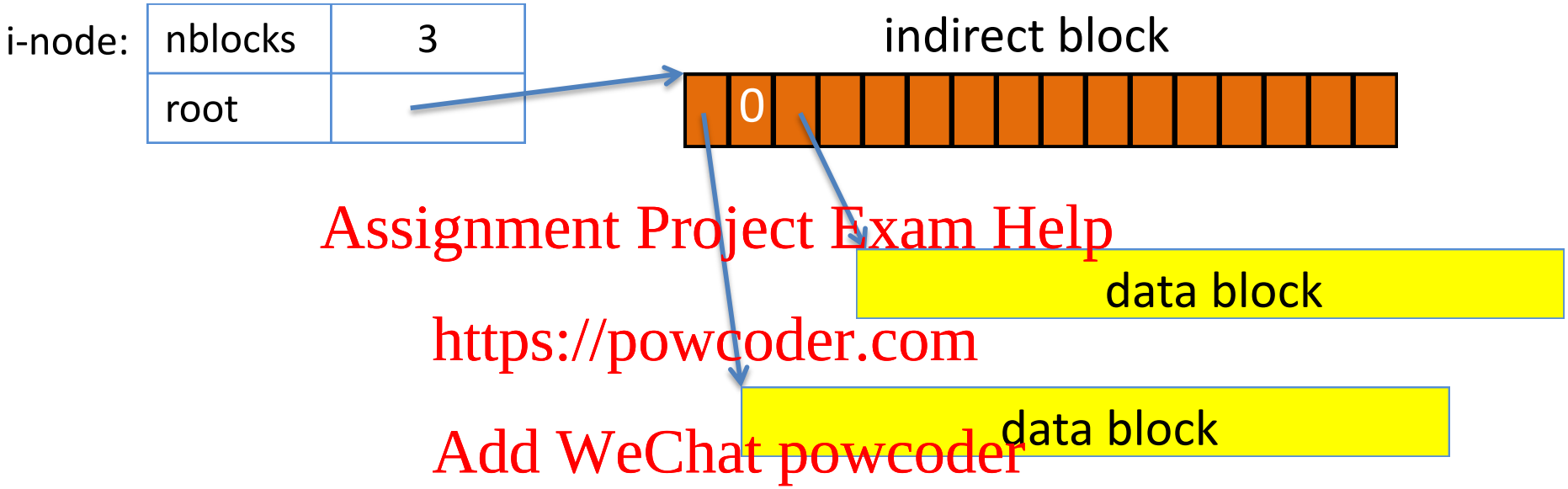
  RPB = REFS_PER_BLOCK

- For example, if rpb = 16:

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

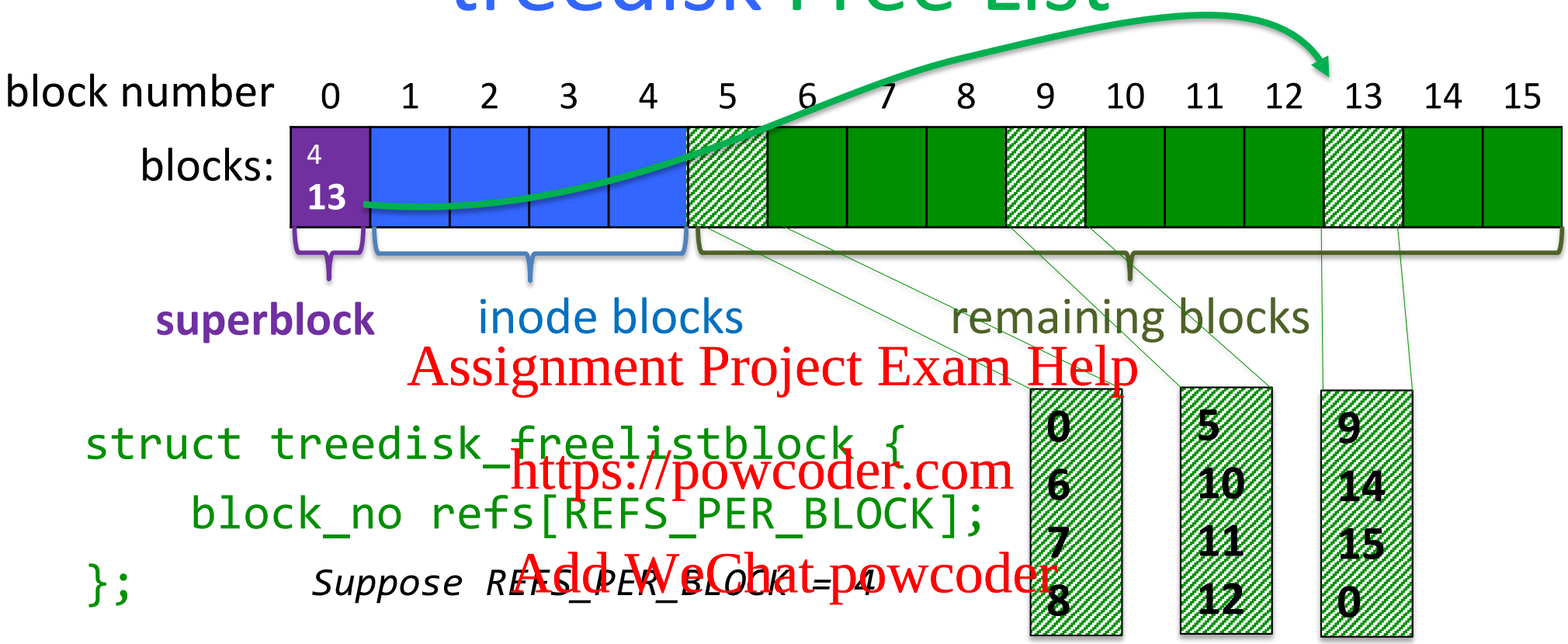
REFS_PER_BLOCK more commonly at least 128 or so

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!

treedisk Free List



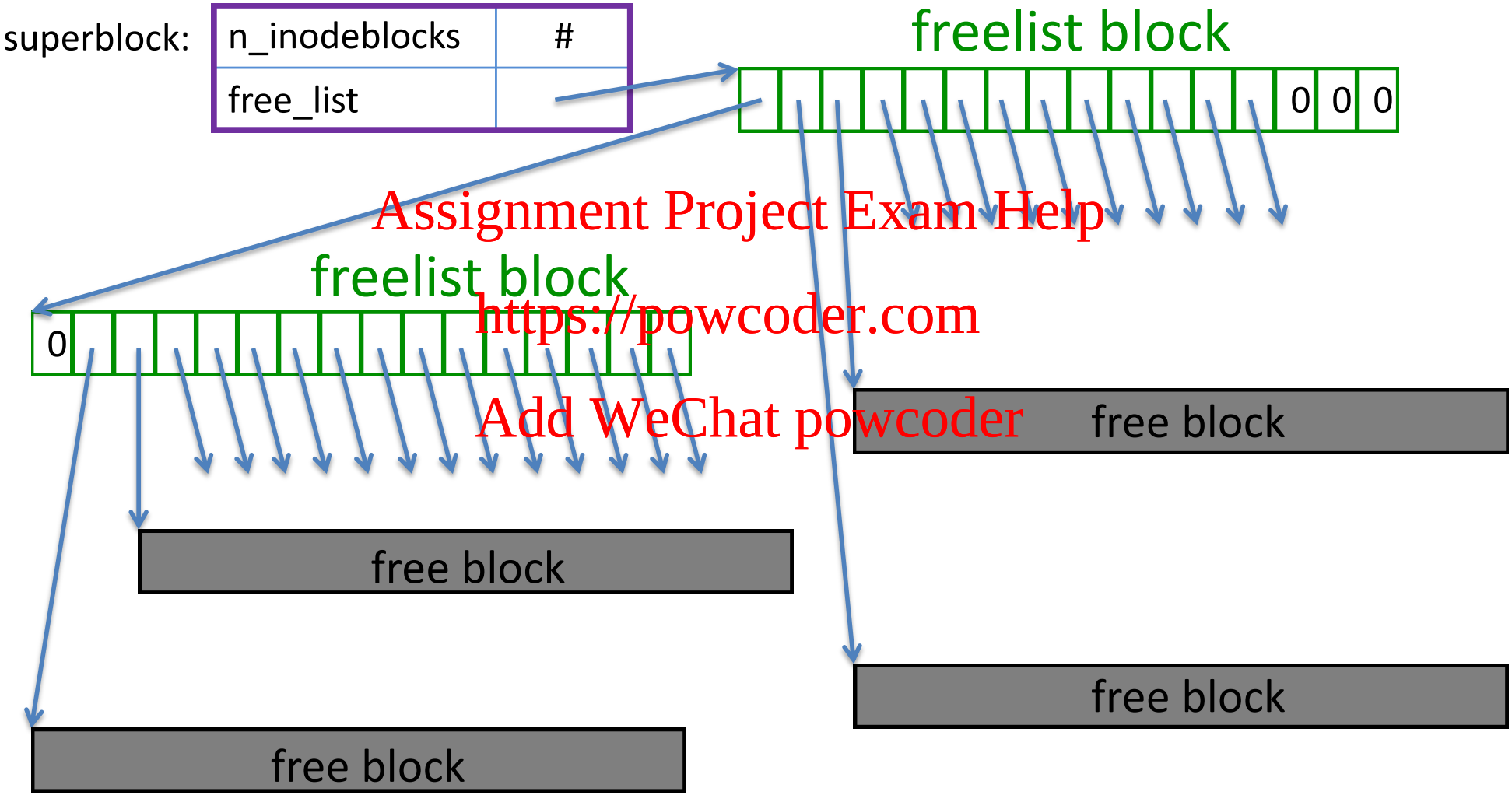

```
struct treedisk_freelistblock {
    block_no refs[REFS_PER_BLOCK];
};
```

Suppose REFS_PER_BLOCK = 4

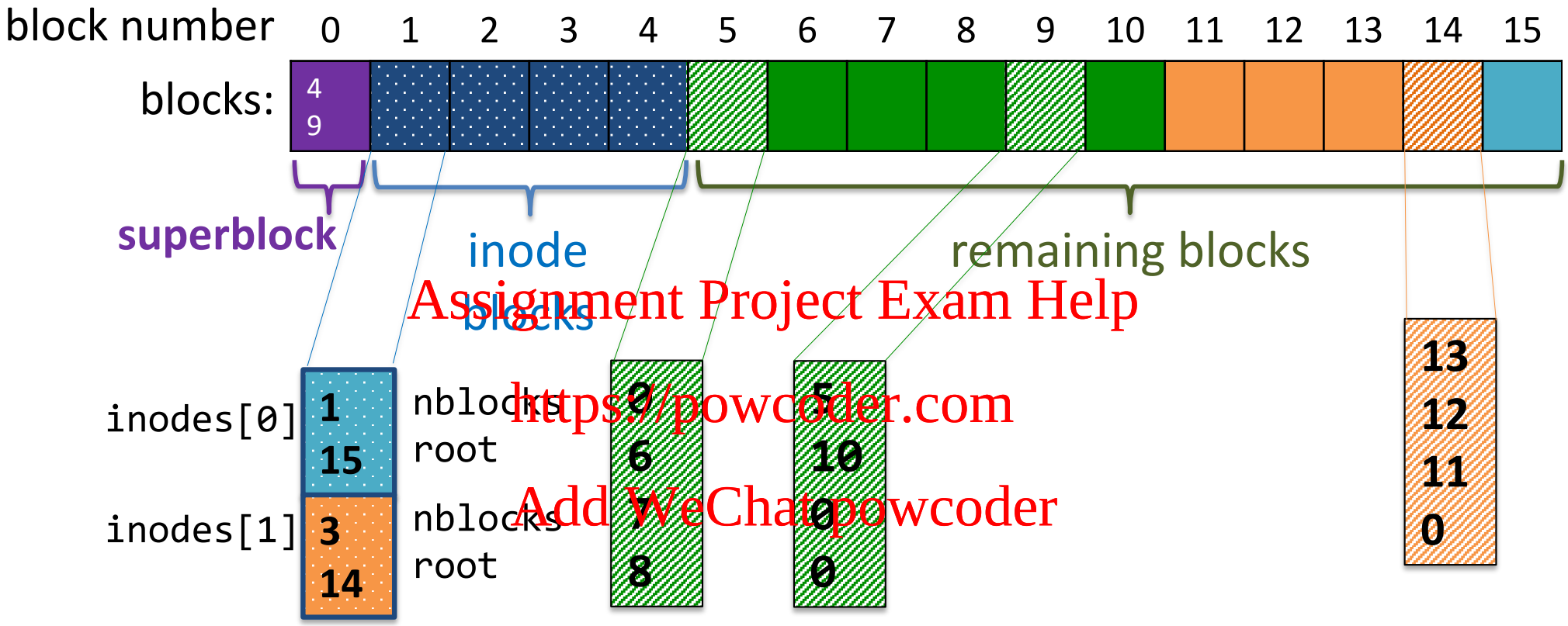
refs[0]: # of another freelistblock or **0 if end of list**

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

treedisk free list



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;
}
```

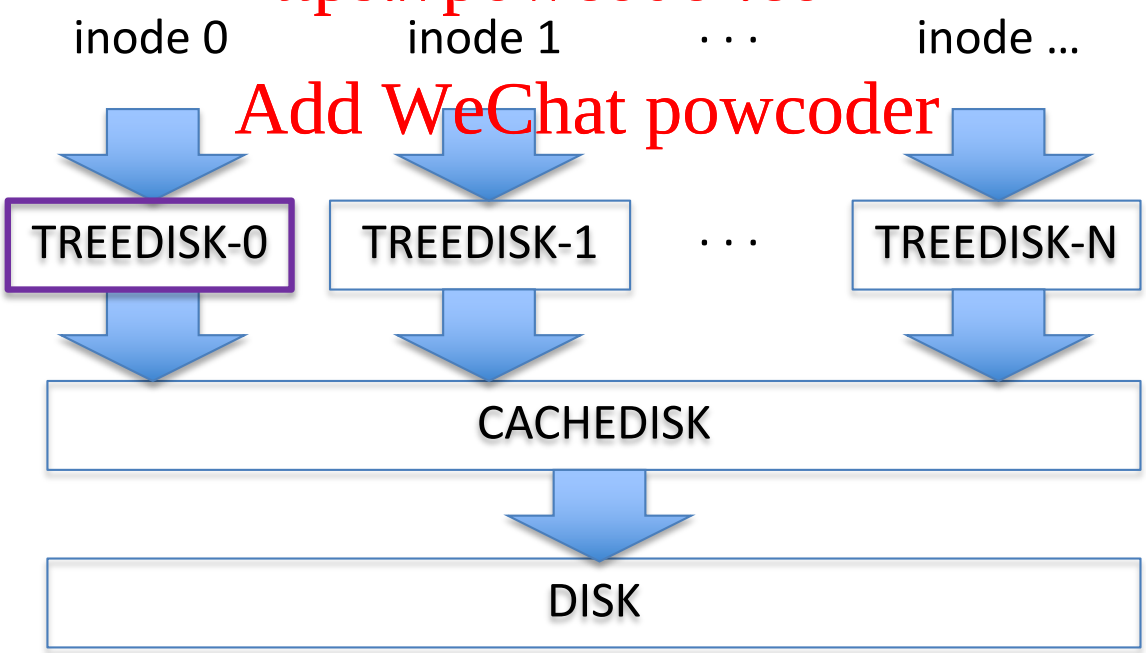
Example code with treedisk

```
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 *file0 = treedisk_init(cdisk, 0);
    block_store_t *file1 = treedisk_init(cdisk, 1);

    block_t block;
    (*file0->read)(file0, 4, &block);
    (*file1->read)(file1, 4, &block);

    (*file0->destroy)(file0);
    (*file1->destroy)(file1);
    (*cdisk->destroy)(cdisk);
    (*disk->destroy)(cdisk);

    return 0;
}
```

Layering on top of treedisk

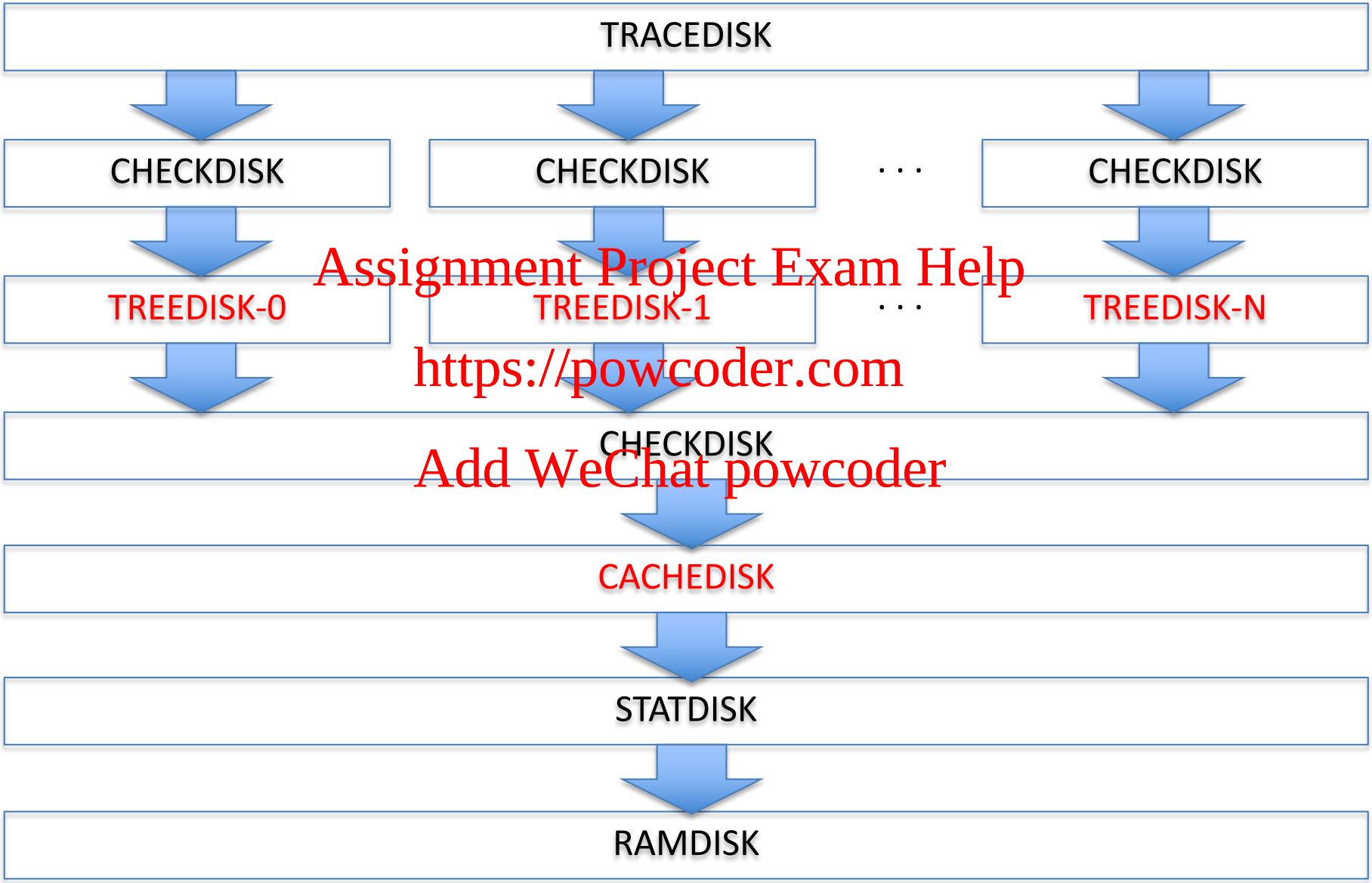
```
block_store_t *treedisk_init(block_store_t *below,
                             unsigned int inode_no);
```

// creates a new file associated with inode_no



individual files

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);
```

Trace file Commands

W:0:3 // write inode 0, block 3

If nothing is known about the file associated with inode 0 prior to this line, by writing to block 3, you are implicitly setting the size of the file to 4 blocks

W:0:4 // write to inode 0, block 4

by the same logic, you now set the size to 5 since you've written to block 4

N:0:2 // checks if inode 0 is of size 2

this will fail b/c the size should be 5

S:1:0 // set size of inode 1 to 0

R:1:1 // read inode 1, block 1

this will fail b/c you're reading past the end of the file (there is no block 1 for the file associated with inode 1, since you just set the size to 0)

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
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
```



```
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

A4: Part 1/3

Implement your own trace file that:

- is at least 10 lines long
- uses all 4 commands (RWNS)
- has an edit distance of at least 6 from the trace we gave you
- is well-formed. For example, it should not try to verify that a file has a size X when the previsus command have in fact determined that it should have size Y. You may find the chktrace.c file useful
- At most: 10,000 commands, 128 inodes, 1<<27 block size

Purpose: convince yourself that your cache is working correctly.

Optional: make a trace that is hard for a caching layer to be effective (random reads/writes) so that it can be used to distinguish good caches from bad ones.

A4: Part 2/3

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 3/3

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!

What to submit

- treedisk.c // with treedisk_setsize(0)
- cachedisk.c
- trace.txt