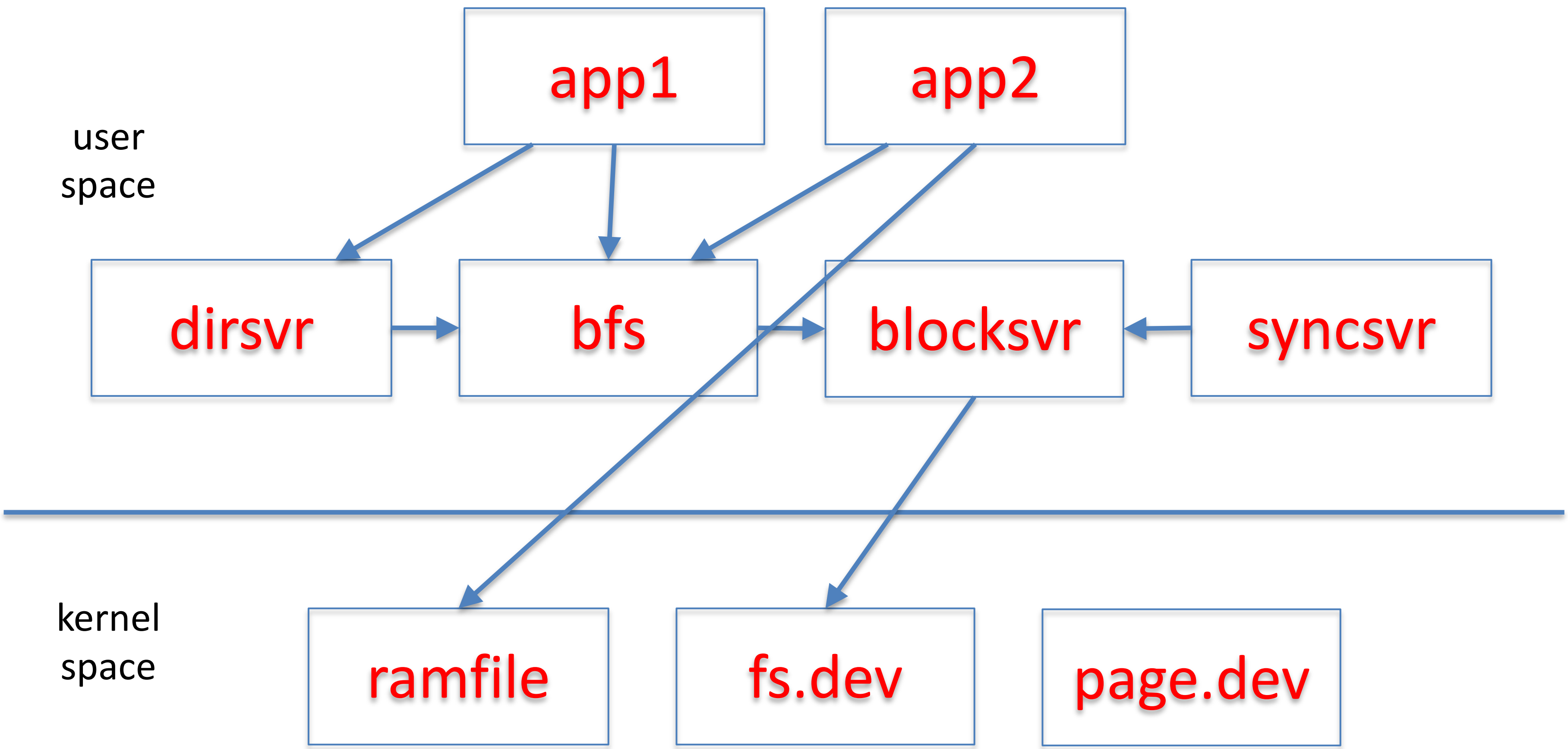
Layered Block-Structured File System & RAID

Robbert 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:
  - a storage object 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, security, etc., just storage

EGOS Storage Architecture



Block Store Abstraction

- A block store consists of a collection of *i-nodes*
- Each i-node is a finite sequence of *blocks*
- Simple interface:
  - block_t block
    - block of size BLOCK_SIZE
  - getninodes() → integer
    - returns the number of i-nodes on this block store
  - getsize(inode number) → integer
    - returns the number of of block on the given inode
  - setsize(inode number, nblocks)
    - set the number of blocks on the given inode
  - release()
    - give up reference to the block store

Block Store Abstraction, cont'd

- read(inode, block number) → block
  - returns the contents of the given block number
- write(inode, block number, block)
  - writes the block contents at the given block number
- sync(inode)
  - make sure all blocks are persistent
    - if inode == -1, then all blocks on all inodes

Simple block stores

- "filedisk": a simulated disk stored on a Posix file
  - block_if bif = filedisk_init(char *filename, int nblocks)
  - has only a single i-node (0)
- "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

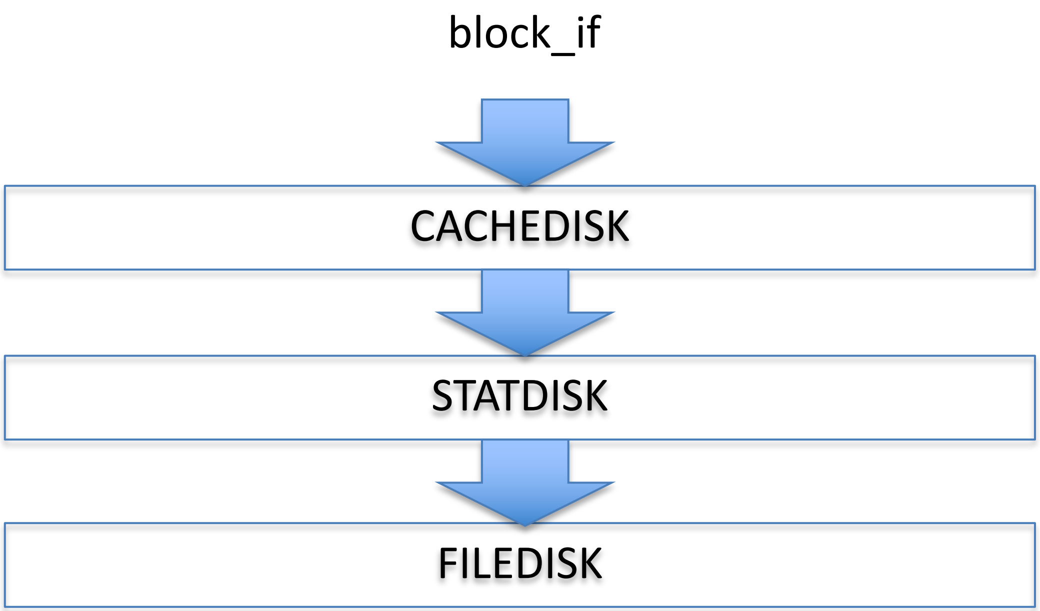
Example code

```
#include ...
#include “egos/block_store.h”

int main(){
   block_if disk = filedisk_init(“disk.dev”, 1024);
   block_t block;
   strcpy(block.bytes, “Hello World”);
   (*disk->write)(disk, 0, 0, &block);
   (*disk->release)(disk);
   return 0;
}
```

Block Stores can be Layered!

Each layer presents a block_if abstraction



keeps a cache of recently used blocks

keeps track of #reads and #writes for statistics

keeps blocks in a Posix file

Example code with layers

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

block_t cache[CACHE_SIZE];

int main(){
    block_if disk = filedisk_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, 0, &block);
    (*cdisk->release)(cdisk);
    (*sdisk->release)(sdisk);
    (*disk->release)(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->getsize = statdisk_nblocks;
    bi->setsize = statdisk_setsize;
    bi->read = statdisk_read;
    bi->write = statdisk_write;
    bi->release = statdisk_release;
    return bi;
}
```

statdisk implementation, cont'd

```
static int statdisk_read(block_if bi, unsigned int ino, 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, unsigned int ino, 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_getsize(block_if bi){ ... }
static int statdisk_setsize(block_if bi, block_no nblocks){ ... }

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

What do we want from storage?

- Fast: data is there when you want it
- Reliable: data fetched is what you stored
- Plenty: there should be lots of it
- Affordable: won't break the bank

Enter: Redundant Array of Inexpensive Disks (RAID)

- In industry, "I" is for "Independent"
- The alternative is SLED, single large expensive disk
- RAID + RAID controller looks just like SLED to computer (*yay, abstraction!*)

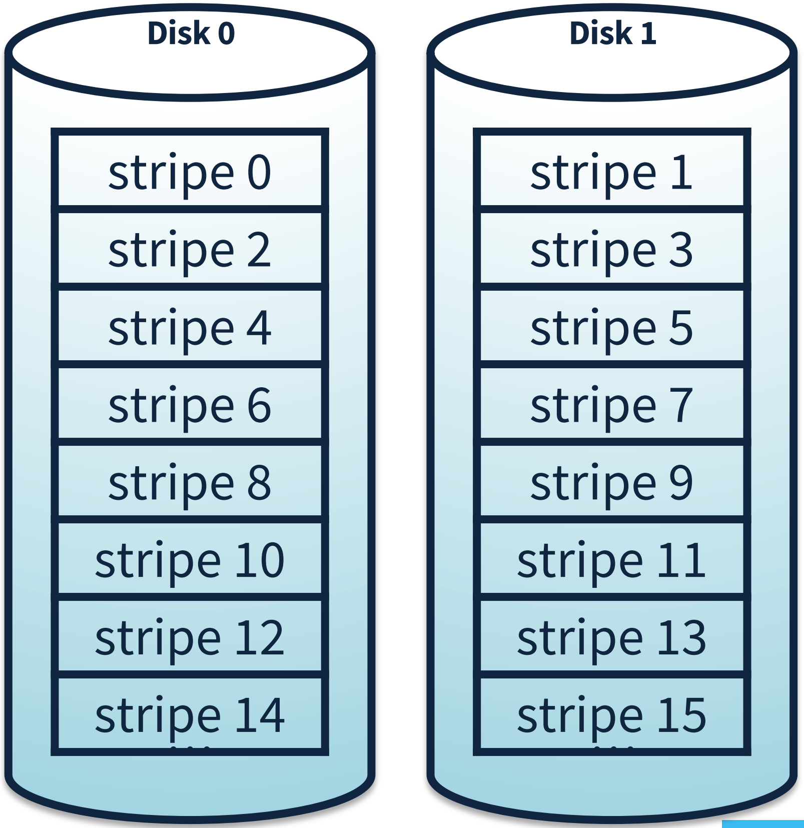
RAID-0

Files striped across disks

+ Fast

+ Cheap

– Unreliable



RAID-1

Disks Mirrored:

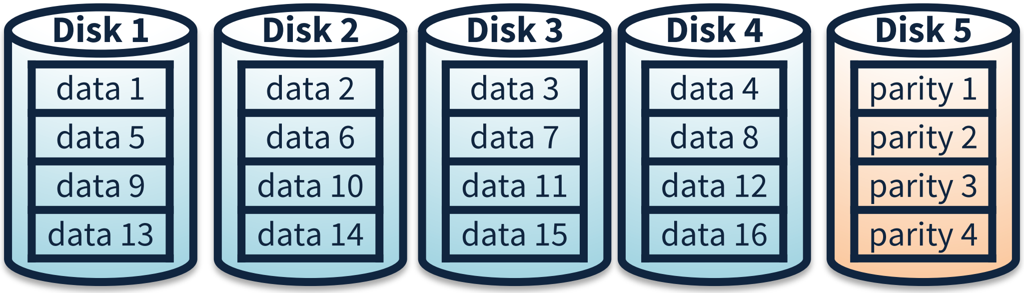
data written in 2 places

+ Reliable
+ Fast

– Expensive

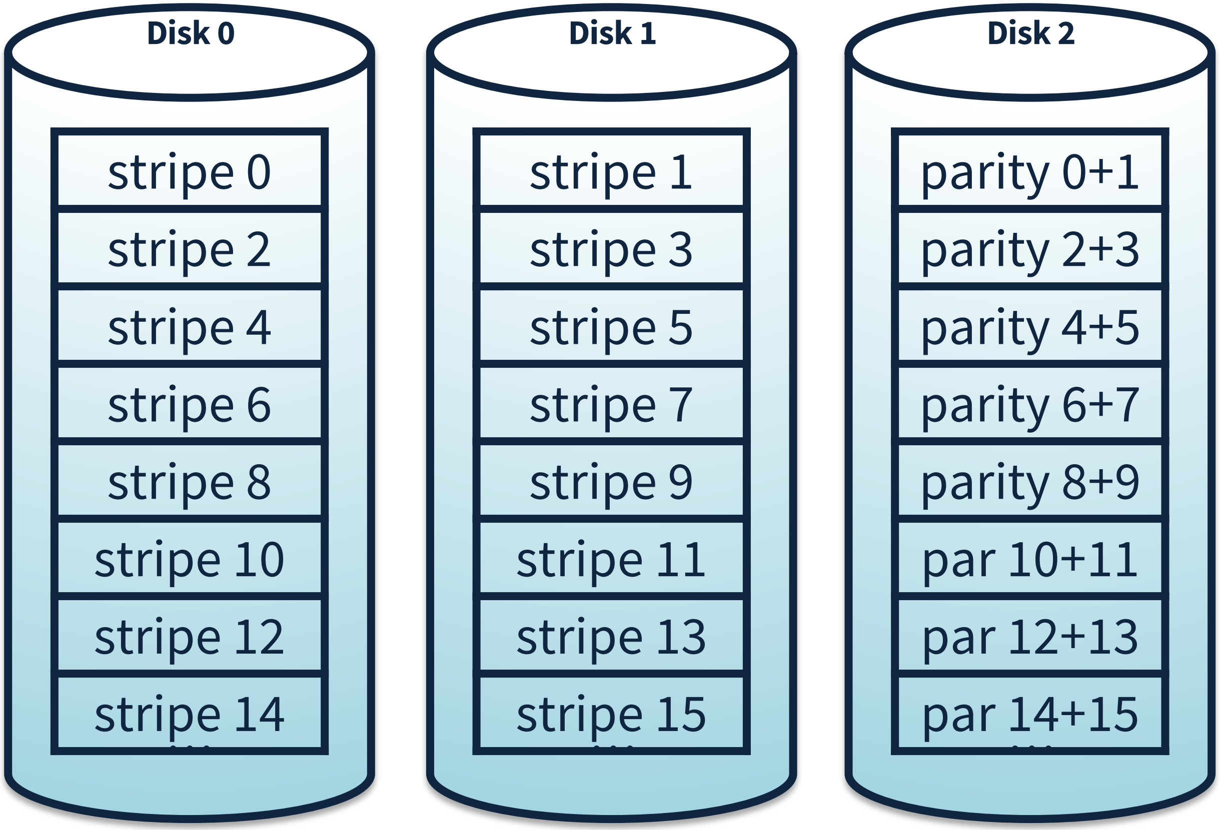
RAID-4

striping + parity disk



RAID-4

striping + parity disk



Using a parity disk

- $D_N = D_1 \oplus D_2 \oplus \ldots \oplus D_{N-1}$
  - $\oplus$ = XOR operation
  - therefore $D_1 \oplus D_2 \oplus \ldots \oplus D_N$ = 0
- If one of $D_1 \ldots D_{N-1}$ fails, we can reconstruct its data by XOR-ing all the remaining drives
  - $D_i = D_1 \oplus \ldots \oplus D_{i-1} \oplus D_{i+1} \oplus \ldots \oplus D_N$

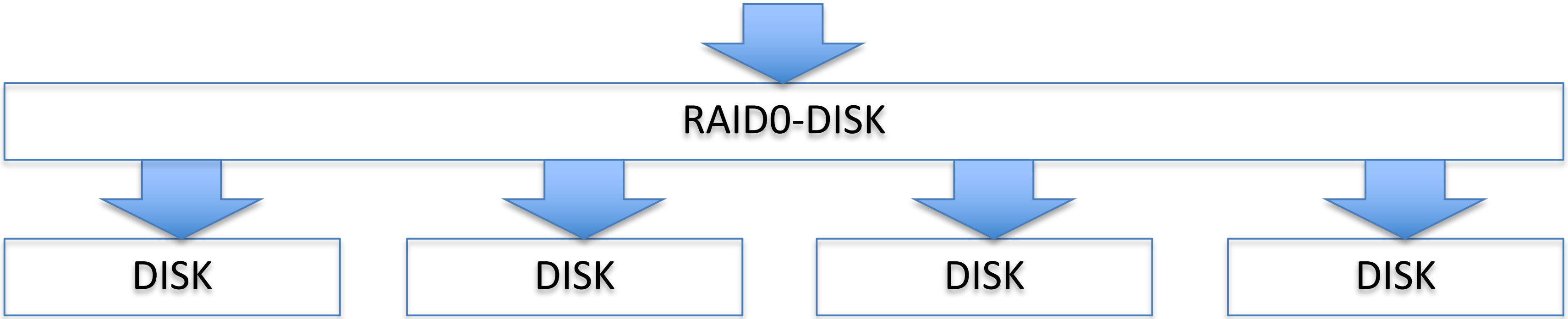
Updating a block in RAID-4

- Suppose block lives on disk D_1
- Method 1:
  - read corresponding blocks on $D_2 \ldots D_{N-1}$
  - XOR all with new content of block
  - write disk D_1 and D_N
- Method 2:
  - read D_1 (old content) and D_N
  - $D_N' = D_N \oplus D_1 \oplus D_1'$

    $= D_1 \oplus D_2 \oplus \ldots \oplus D_{N-1} \oplus D_1 \oplus D_1'$

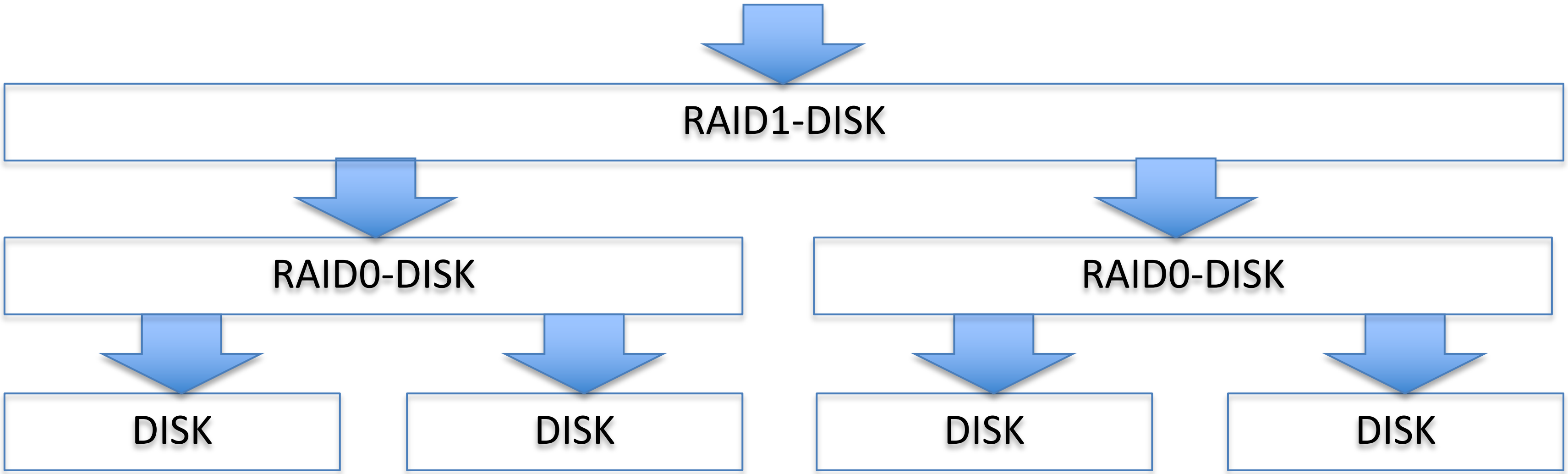
    $= D_1' \oplus D_2 \oplus \ldots \oplus D_{N-1}$
  - write disk D_1 and D_N

RAID 0



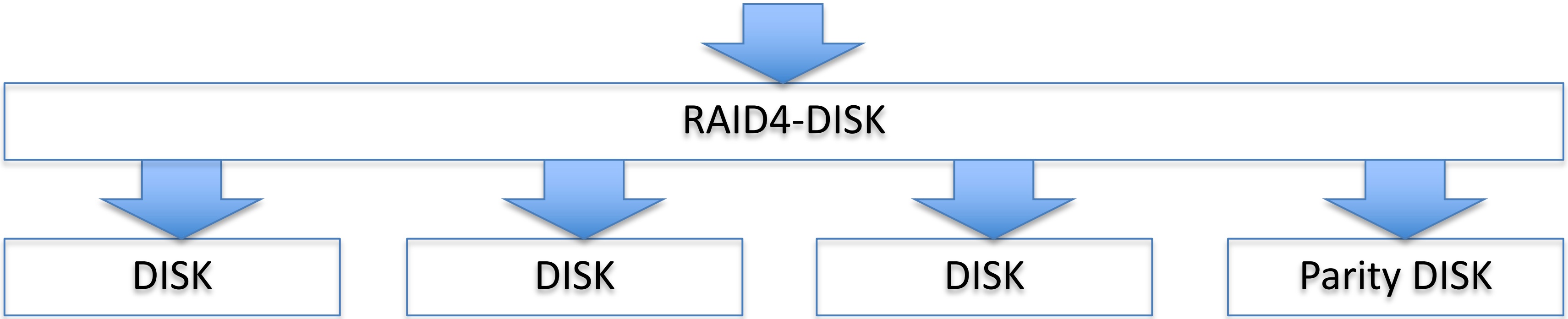

```
block_if raid0disk_init(block_if *below, unsigned int nbelow);
```

RAID 0+1




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

P3: write raid4disk.c




```
block_if raid4disk_init(block_if *below, unsigned int nbelow);
```

Careful: beware of corner cases!

- read when one of the disks have failed
- write when one of the disks have failed
  - could be data disk or parity disk!