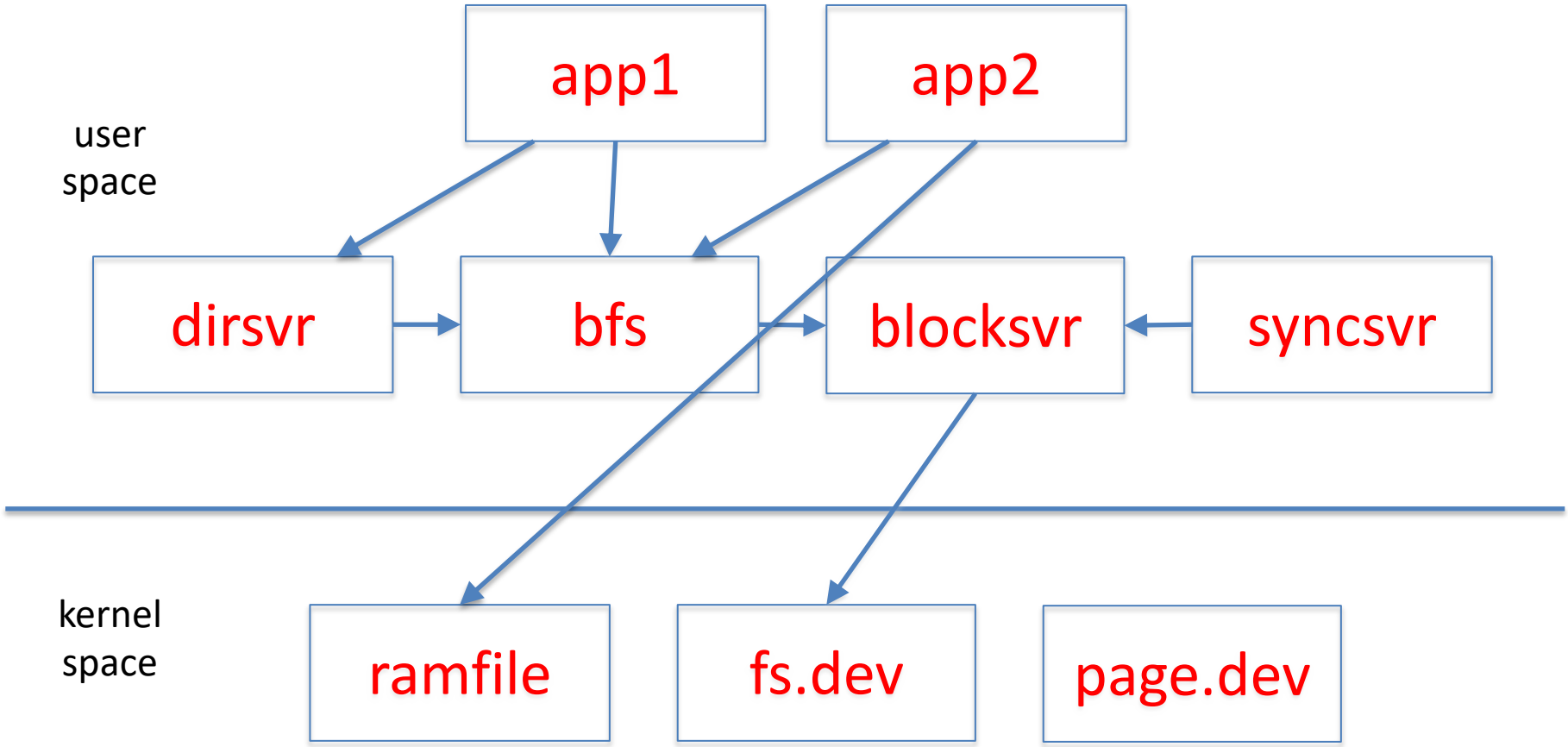
Layered Block-Structured File System

Robbert van Renesse

Intro

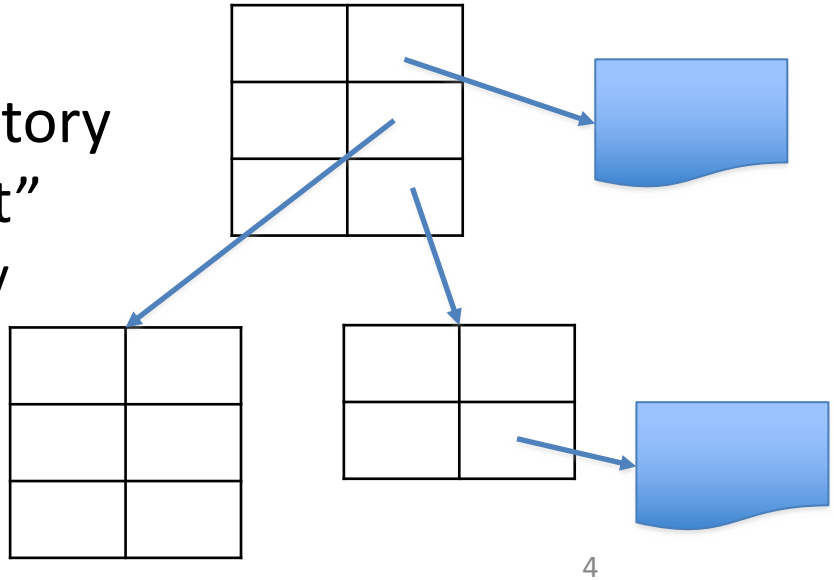
- Underneath any file system, database system, etc. there are one or more *block stores*
- A block store provides a disk-like interface:
  - a storage object is a sequence of blocks
    - typically, a few kilobytes
  - you can read or write a block at a time
- The block store abstraction doesn't deal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

EGOS Storage Architecture



dirsvr: directory server

- Maps path names to file identifiers
  - A file identifier is a pair (process id, i-node number)
- Each directory is a file that maintains an array of simple-name → file identifier mappings
  - e.g., { x.txt: 9:34, y.dir: 6:54, z.exe: 9:4 }
- Directories can be organized into graphs (usually trees)
- Root directory is global
- Each process has a working directory
- Can recursively resolve "a/b/x.txt"
  - looks up a.dir in working directory
  - looks up b.dir in a
  - looks up x.txt in b



bfs: block file server

- Stores all its user and meta data in blocksvr
- Maintains for each file a “stat structure”:
  - size in bytes
  - owner
  - modification time
  - access control information
  - etc.
- files are indexed by i-node numbers
  - 0, 1, 2, …
  - #i-nodes determined by blocksvr

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

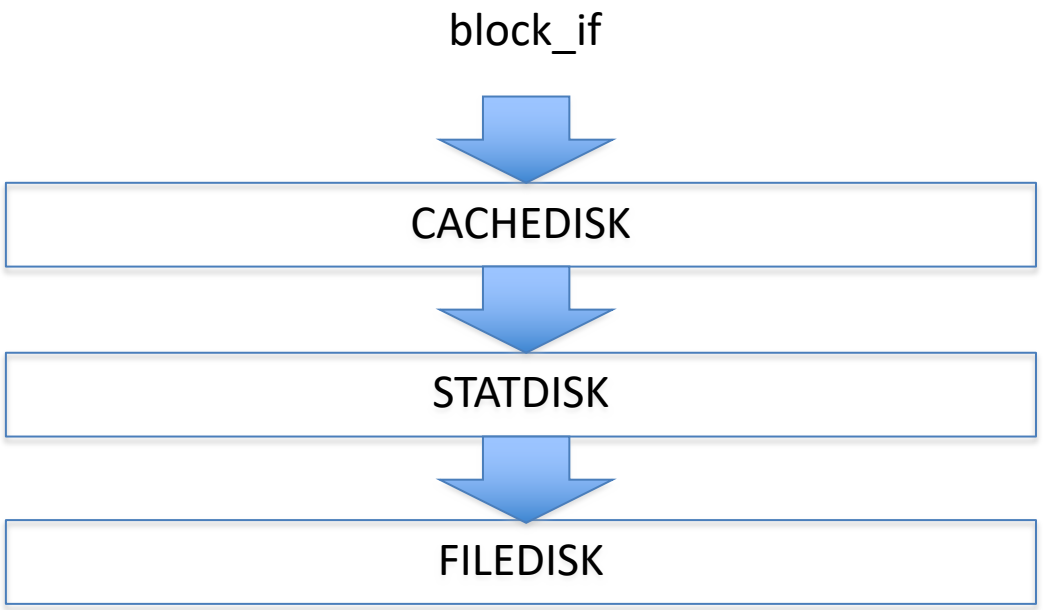
- "filedisk": a simulated disk stored on a Posix file
  - block_if bif = filedisk_init(char *filename, int nblocks)
  - has only a single i-node (0)
- "ramdisk": a simulated disk in memory
  - block_if bif = ramdisk_init(block_t *blocks, nblocks)
    - Fast but volatile
- block_if is a pointer to the 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

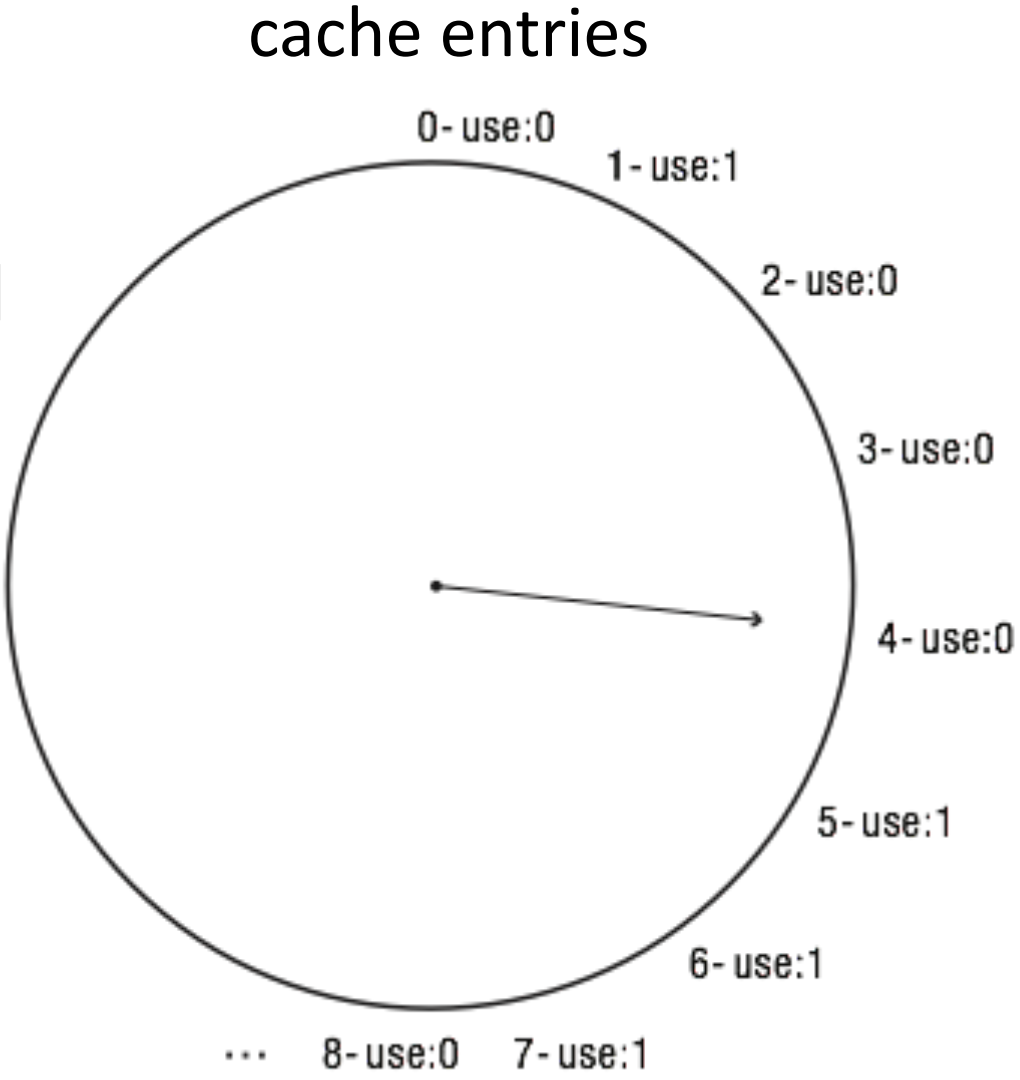
```
static int statdisk_read(block_if bi, unsigned int ino, block_no offset,
block_t *block){
    struct statdisk_state *sds = bi->state;
    sds->nread++;
    return (*sds->below->read)(sds->below, offset, block);
}

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);
}
```

P3: Implement a cache layer

- Suggested: based on clock algorithm
- Two versions:
  1. write-through
  2. write-behind *aka* write-back
- Tricky part: what to do if cache is full?

Clock Algorithm

- To allocate a block, inspect the *use* bit in the PTE at clock hand and advance clock hand
- Used? Clear *use* bit and repeat



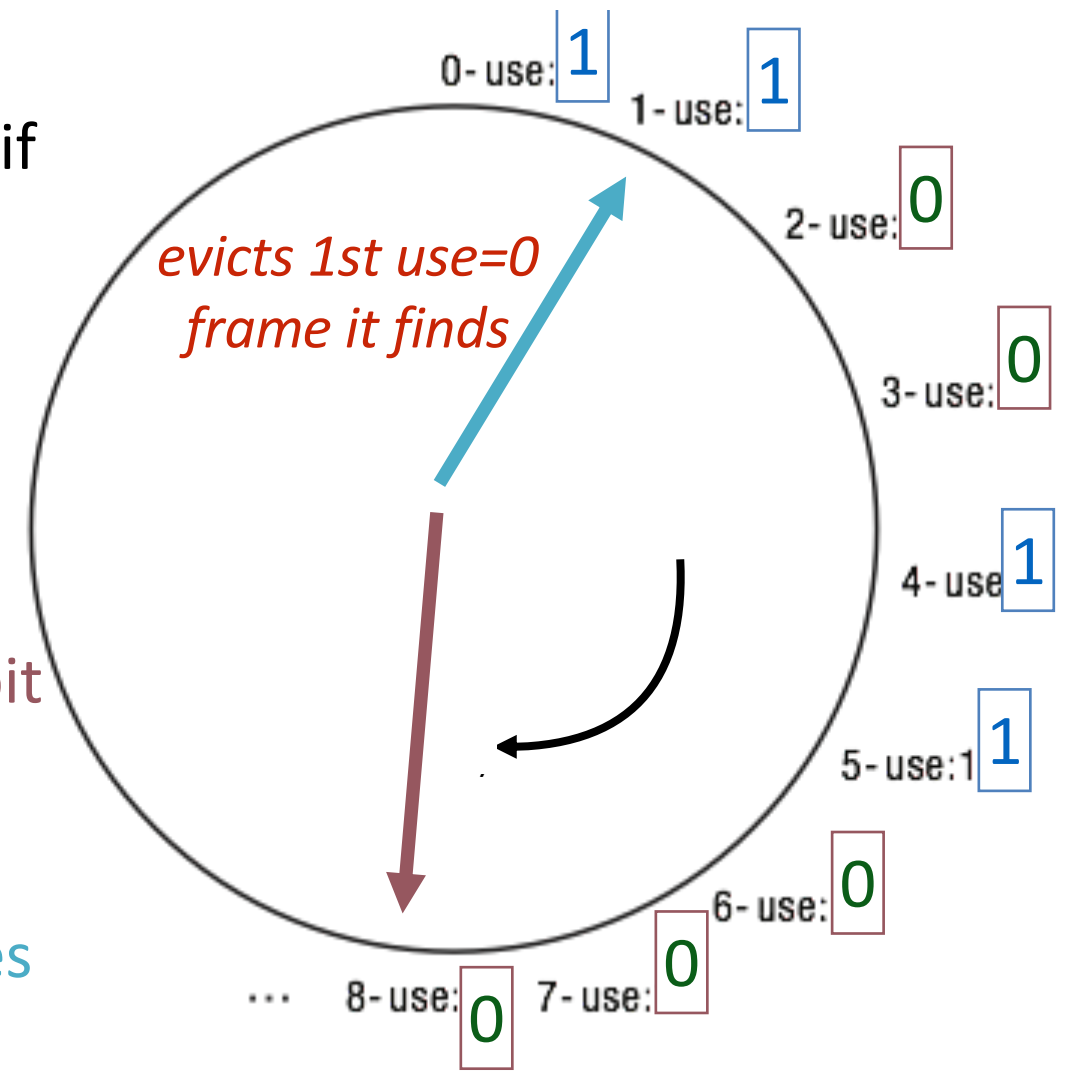
Two-Handed Clock

- One-handed clock: What if #blocks is very large?

- Use two hands!
- (at fixed angle)

- Leading hand clears *use* bit
- slowly clears history
- finds victim candidates
- Trailing hand evicts frames
- with *use* bit set to 0

- Big angle? Small angle?



blue 1's were referenced after use bit was cleared by green hand