WRITING LINUX FS

4 FUN

Outline

- Why
- Main Concepts and bit of history
  - *Earlier design decisions*
  - *On disk layout*
- Implementing own FS
- On disk layout
- Code Fragments:
  - *Kernel Implementation*
  - *Other tools mkfs, fsdb*

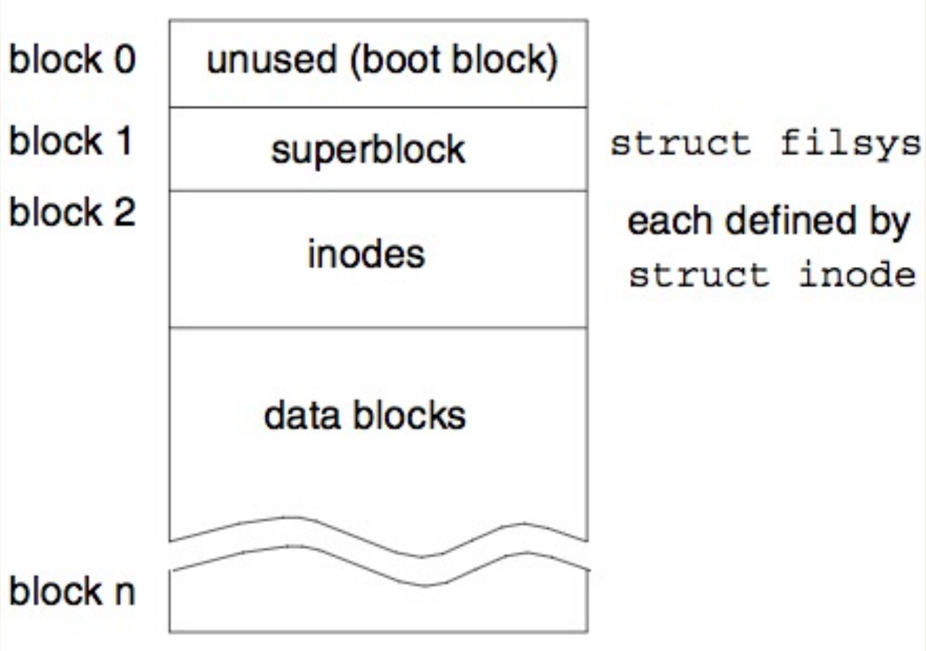
Why this talk?!

- **Cons**
  - *Writing FS is quite time consuming (approx. 10 years…)*
  - *Just few production ready FS, many abandoned or not truly maintained*

- **Pros**
  - ***Learning: Address specific gap***
  - ***Solving other complicated problems***
    - Storage stack is complicated and usually became a bottleneck
    - Data is foundation of most todays application

Early days: 6^{th} ed. of UNIX

- File system: one internal component of the kernel
- Not possible to use other FS
- Block size as fixed 512 bytes
- Possible indirect block (up to 3 level depth)
- Max size of file: 32*32*32 data blocks



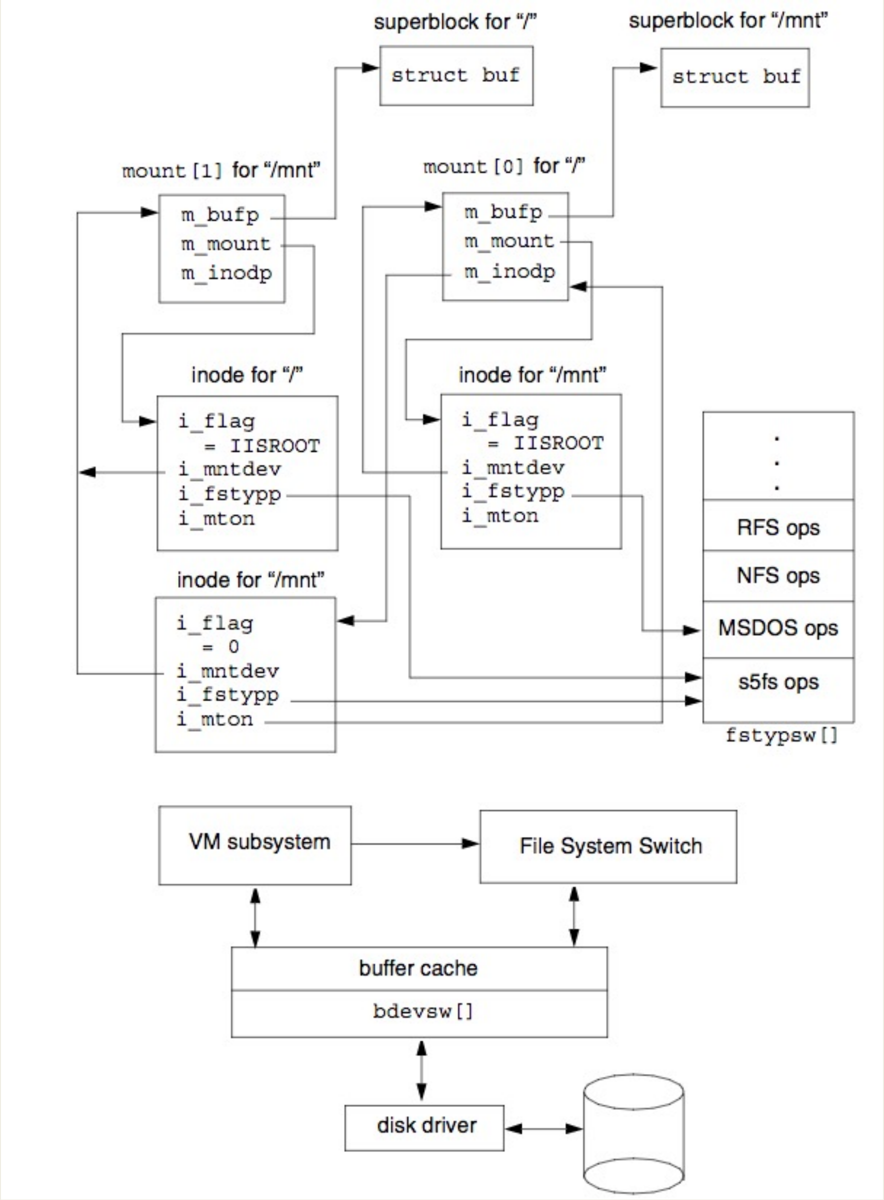
Early days: 6th ed. of UNIX

```
struct inode  {
    i_mode    // file type*
    i_nlink   // nr of hard links
    i_uid
    i_gid
    i_size
    i_addr[7] // 7 pointers to blocks
    i_mtime   // modify time
    i_atime   // access time
}
```

Note: Mode define specified file: directory IFDIR, block device IFBLK or char dev IFCHR

File System Switch

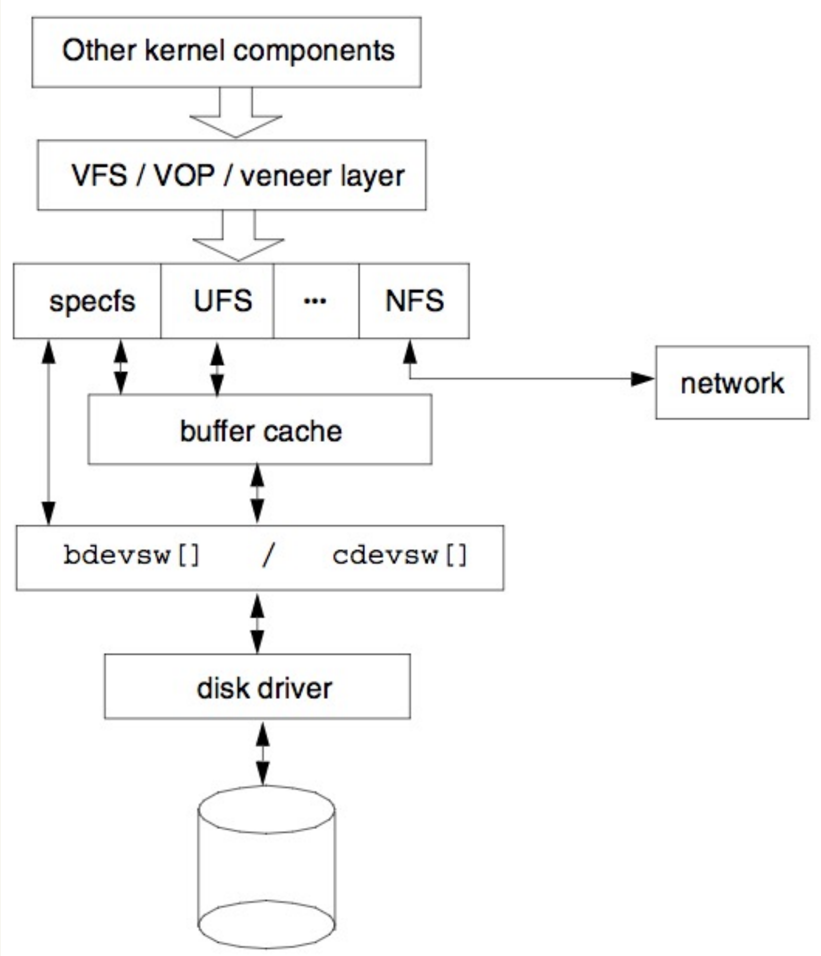
- Main goal: provide framework under which multiple filesystems could exist in parallel
- Divide FS to independent layer and in-core (FS dependent)
- FS representation for file called “inode”
- **Short lived, being replaced by Sun VFS.**



SunOS VFS/vnode

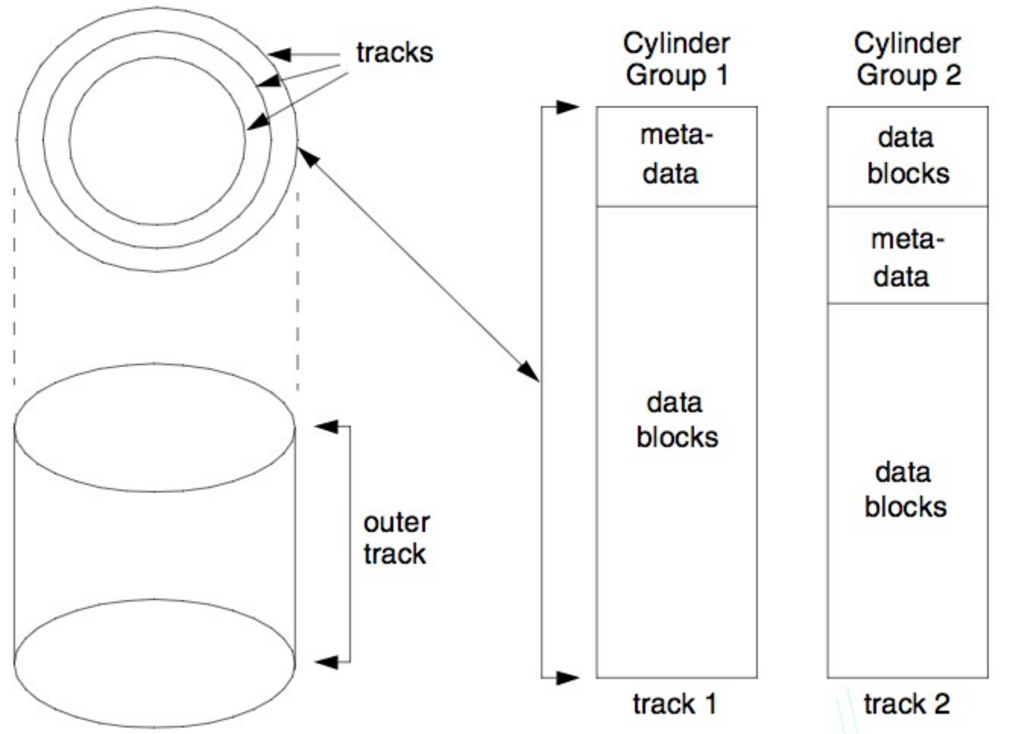
- VFS unified UNIX filesystems by split into independent and in-core layers
- vnodes are part of VFS and inodes part of the in-core layer
- Common layer for kernel components to r/w to the files
- vnode contain private data field which was used to store in-core inode

```
inode->i_private = dm_inode
```



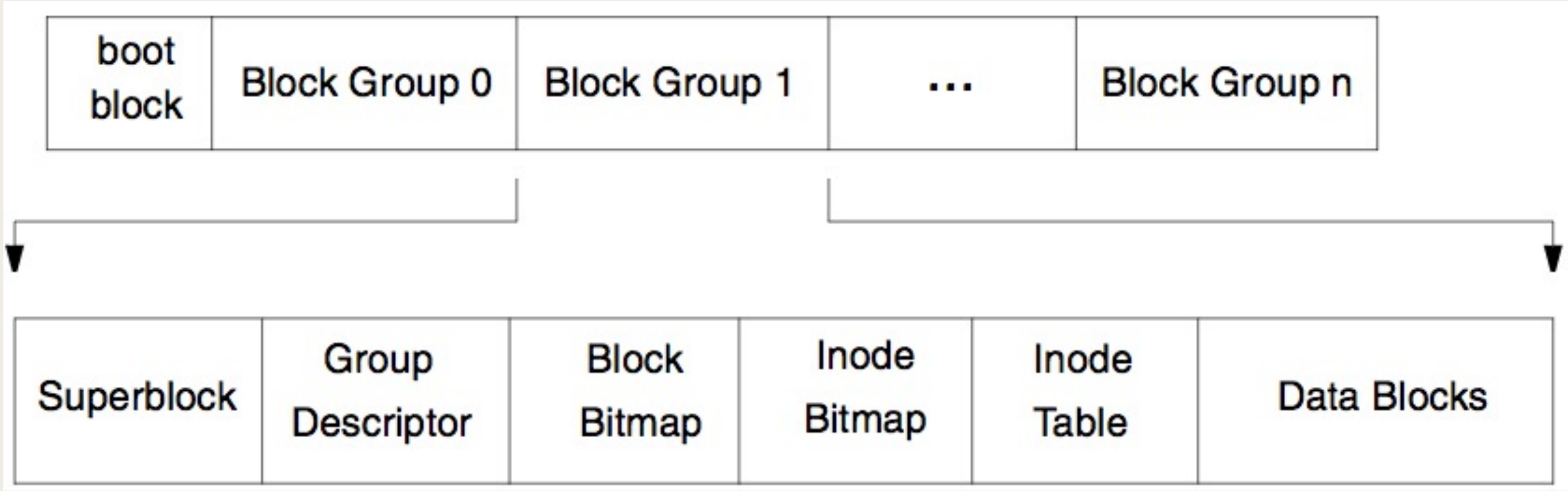
On Disk Layout: UFS

- Initial UNIX FS has poor performance
- UFS new design concerned the layout of data on disks i.e:
  - *Track contains same amount of data*
- The old UNIX FS was only able to use 3 to 5 percent of the disk bandwidth while the FFS up to 47 percent of the disk bandwidth*



On Disk layout: EXT2

- EXT2 divide filesystem to number of block groups
- inode allocation done during mkfs
- Fixed offset for first Block Group, space for bootloader

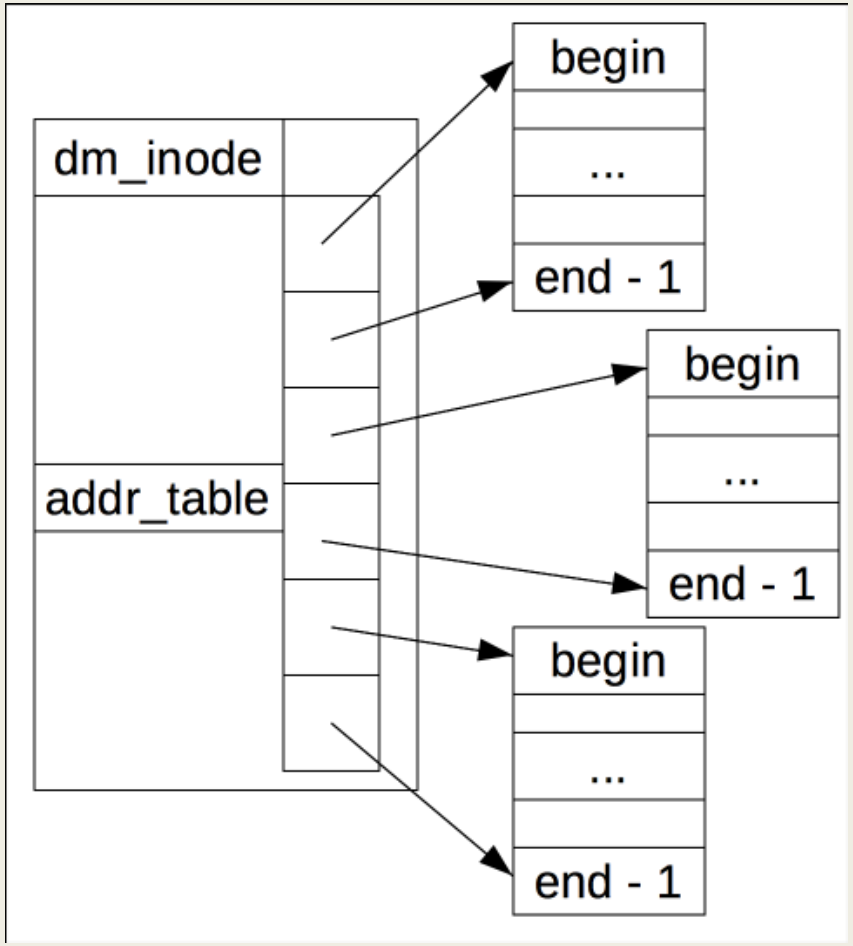


Sample implementation dummyfs

- Implemented as in kernel module (possible to implement in user space using FuseFS)
- Provide basic functionality necessary to mount read write files to the disk
- *Pseudo* modern on disk layout
- Good starting point to learn internal/implementing more advanced features
- Not using kernel caching mechanisms

Inode structures:

- inode has addr_table which describe 3 possible extends
- Extends are contiguous space of block described by range Begin-End
- Default size of range during allocation



On Disk layout

- Simple but not trivial
- Inode table and Inode bitmap are ‘files’ which allow them to scale
- Blocks addresses are 32 bit integers which define limits

Boot offset	Superblock	Object table	Inode table	Inode bitmap	root inode	Data Blocks

Basic components:

- Main implementation of specific components in
  - *dir.c*
  - *file.c*
  - *inode.c*
  - *super.c*
- Structures inside dummy_fs.h shared by kernel and user space components
- Module implementation in dummyfs.c: registration of FS, allocate memory for inodes

In-Core structures

```
struct dm_inode {
    u8      i_version;
    u8      i_flags;
    u32     i_mode;
    u32     i_ino;
    u16     i_uid;
    u32     i_ctime;
    u32     i_mtime;
    u32     i_size;
    u32     i_addrb[DM_EXT_SIZE];
    u32     i_addre[DM_EXT_SIZE];
};
```

```
struct dm_superblock {
    u32   s_magic;
    u32   s_version;
    u32   s_blocksize;
    u32   inode_table;
    u32   inode_cnt;
    u32   inode_bitmap;
};

struct dm_dir_entry {
    u32 inode_nr;
    u8 name_len;
    char name[256];
};
```

Fragments: **Mount**

```
// mount process:
struct file_system_type
dummyfs_type = {
    .name = "dummyfs",
    .mount = dummyfs_mount,
    .kill_sb = dummyfs_kill_sb,
    .fs_flags = FS_REQUIRES_DEV
};

register_filesystem(&dummyfs_type)
```

```
struct dentry *dummyfs_mount(…)
*fs_type, flags, *dev_name, *data
{
mount_bdev(fs_type, flags, dev_name, data,
         dummyfs_fill_super);
...

static int dummyfs_fill_super(...)
*sb, *data, silent
{
    struct dm_superblock *d_sb;
    struct buffer_head *bh;
    struct inode *root_inode;
    struct dm_inode *root_dminode;

    bh = sb_bread(sb, DM_SUPER_OFFSET);
    d_sb = (struct dm_superblock *)bh->b_data;

    bh = sb_bread(sb, DM_ROOT_INODE_OFFSET);
    root_dminode = (struct dm_inode *)bh->b_data;
    root_inode = new_inode(sb);
...
```

Fragments: **lookup**

“implement ls”

```
// file ops
const struct file_operations
dummy_dir_ops = {
    .iterate_shared = dummy_readdir,
};
const struct file_operations
dummy_file_ops = {
    .read_iter = dummy_read,
    .write_iter = dummy_write,
}

// Ops filled during the iget(inode_nr)
// ls from user space will call readdir
// for inode
```

```
int dummy_readdir(struct file *filp, struct dir_context
*ctx)
... // get from filp underlaying inode
/* For each extends from file */
for (i = 0; i < DM_INODE_TSIZE; ++i) {
    u32 blk = di->i_addrb[i], e = di->i_addre[i];

    while (blk < e) {

        bh = sb_bread(sb, blk);
        BUG_ON(!bh);
        dir_rec = (struct dm_dir_entry *)(bh->b_data);

        for (j = 0; j < sb->s_blocksize; j+=size(*dir_rec))
{
            /* skip empty/free inodes */
            if (dir_rec->inode_nr == 0xdeeddeed)
                skip;

            dir_emit(ctx, dir_rec->name,
                     dir_rec->name_len,
                     dir_rec->inode_nr,
                     DT_UNKNOWN);
            filp->f_pos += sizeof(*dir_rec);
            ctx->pos += sizeof(*dir_rec);
            dir_rec++;
        }

        /* Move to another block */
        blk++;
        bforget(bh);
    }
}
```

Fragment read/write

```
// file ops
const struct file_operations
dummy_file_ops = {
    .read_iter = dummy_read,
    .write_iter = dummy_write,
}

// Ops filled during the iget()
// ls from user space will call readdir
// for inode
```

```
ssize_t dummy_write(struct kiocb *iocb, struct
iov_iter *from)
...
    //Get VFS and in-core structures from io
    inode = iocb->ki_filp->f_path.dentry->d_inode;
    sb = inode->i_sb;
    dinode = inode->i_private;
    dsb = sb->s_fs_info;

    // Find the block and offset to write
    blk = dm_alloc_ifn(dsb, dinode, off, count);
    boff = dm_get_loffset(dinode, off);

    bh = sb_bread(sb, blk);

    buffer = (char *)bh->b_data + boff;
    copy_from_user(buffer, buf, count);
    iocb->ki_pos += count;

    mark_buffer_dirty(bh);
    sync_dirty_buffer(bh);
    brelse(bh);

    store_dmfs_inode(sb, dinode);

    return count;
}
```

User Space tools

- **mkfs**
  - *Initialize the device to be used by FS.*
  - *Write initial FS state*
- **fsdb**
  - *Development tool reading structures from raw device*
  - *Understand on disk structure*
- **fsck**
  - *Try to recover inconsistent state of FS (due to crash/corruption).*

Fragment mkfs

```
// Write initial FS state to the device
// arg is targ device: /dev/sdb or lv /dev/sdb1
fd = open(argv[1], O_RDWR);
if (fd == -1) {
    perror("Error: cannot open the device!\n");
    return -1;
}
// wipe out device before writing
wipe_out_device(fd, 1));
// Write actual on disk structure
write_superblock(fd));
write_metadata(fd);
write_inode_table(fd);
write_root_inode(fd);
write_lostfound_inode(fd);

//write entries to inode table
write_root2itable(fd);
write_laf2itable(fd);
```

```
int write_root_inode (int fd) {

  // construct root inode
  struct dm_inode root_inode = {
    .i_version = 1,
    .i_flags = 0,
    .i_mode = S_IFDIR | S_IRWXU | S_IROTH | S_IXOTH,
    .i_uid = 0,
    .i_ctime = dm_ctime,
    .i_mtime = dm_ctime,
    .i_size = 0,
    .i_ino = DM_ROOT_INO,
    .i_addrb = {DM_ROOT_INODE_OFFSET + 1, 0, 0},
    .i_addre = {DM_ROOT_INODE_OFFSET + DM_EXALLOC+1, 0,
0},
  };

  lseek(fd, DM_ROOT_OFFSET * DM_BSIZE, SEEK_SET);
  write(fd, &root_inode, sizeof(root_inode)));

  // write root to the inode table as a first entry
  lseek(fd, (DM_ITABLE_OFFSET + 1) * DM_BSIZE,
SEEK_SET);
  write(fd, &blk, sizeof(uint32_t))
```

Other resources:

- J.Lions: "A commentary on the sixth edition UNIX Operating System”
- V6 sources: https://minnie.tuhs.org/cgi-bin/utree.pl
- S.R. Kleiman (86): “Vnodes: An Architecture for Multiple File System Types in Sun UNIX”
- McKusick (84): “A Fast File System for UNIX.”
- Steve D. Pate: "UNIX Filesystems: Evolution, Design and Implementation”
- github.com/gotoco/dummyfs

Q&A