CSE134 Assignment 3 Design Doc

Group ABAJ – Alexander Sung, Benson Ho, Aidan Sojourner, James Quiaoit

May 24, 2021

1 Introduction

This is the design document for Assignment 3 (`kvfs`) for CSE 134, Spring 2021, at the University of California, Santa Cruz.

1.1 Goals

`kvfs` is an in-kernel filesystem for FreeBSD that works like a key-value store.

- Each file in `kvfs` is uniquely represented as a 160-bit numeric key, shown to the user as 40 hexadecimal characters.
- Each "key" is associated with a single 4KiB "value", which is initialized to all 0s when the key-value pair is first created.
- Keys can be created, retrieved, renamed, updated, and removed using standard UNIX file operations.
- Every file in `kvfs` also contains additional metadata, namely a 64-bit modification time and a 16-bit reference count (currently unused).
- `kvfs` contains no directory hierarchy – each key-value pair (file) in `kvfs` is stored directly under the filesystem root.
- No access control or permissions are implemented.
- `kvfs` supports a maximum of 2^{30} files. This corresponds to a maximum partition size of 4TiB.

The program `mkkvfs` is also provided, which allows the user to format a disk for use with the `kvfs` filesystem.

2 Design

2.1 Disk Layout

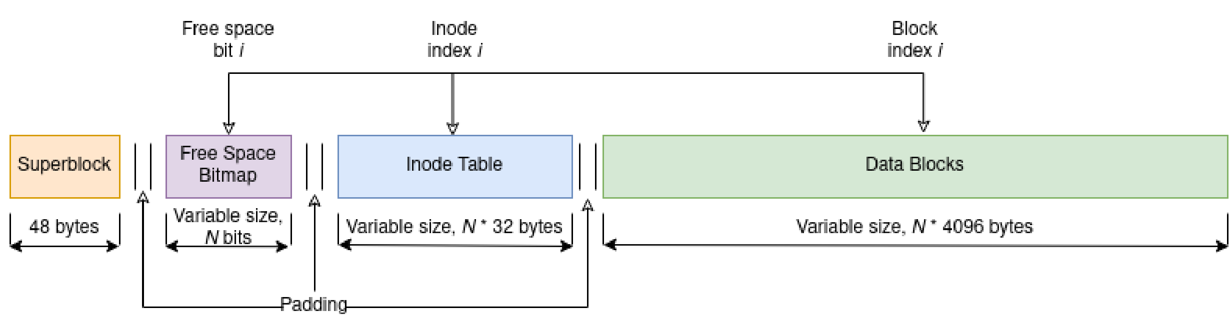

Figure 1: `kvfs` Disk Layout

The filesystem "superblock" is allocated at the very beginning of the disk. It contains metadata about the filesystem, and is described in detail in the Superblock section.

Each file in `kvfs` is represented by a pair of (inode, block), where the inode contains metadata about the file, and the block contains the contents. Because files in `kvfs` have a fixed 4KiB size, each inode has one and only one data block associated with it. A fixed number of inodes (`struct kvfs_inode`) are allocated at a known location in the disk. Each inode is indexed by a unique offset, which also corresponds to the index of the data block for that file.

The contents of each inode are described in the Inode section.

Additionally, a bit-map of free inode/data block pairs is also placed just after the superblock. For details, see the Free List section.

2.1.1 Superblock

The `kvfs` superblock on-disk contains the following entries:

```
struct kvfs_superblock {
    uint16_t magicnum;          /* kvfs magic number */
    uint16_t superblock_size;   /* size of superblock on disk */
    off_t freelist_off;         /* block offset of free list */
    off_t inode_off;            /* block offset of inode allocation table */
    off_t data_off;             /* block offset of data blocks */
                                /* supports a maximum of 2^30 blocks */
    uint32_t block_count;       /* number of data blocks in this filesystem. */

    uint64_t flags;             /* filesystem flags */
    uint64_t fs_size;           /* actual filesystem size in bytes */
};
```

The superblock contains only static data, and is never written to after first being created in `mkkvfs`. Additionally, the `flags` variable is currently unused, being reserved for future use.

2.1.2 Free List

Each data block/inode pair in `kvfs` is either free or in-use. We represent this "free list" on-disk with a bitmap. For each key-value pair which can be stored on disk, one bit is set to indicate whether an inode and data block pair at that block index has been allocated or not. A bit value of `0` means that the block is free, and a value of `1` means that the block is in-use.

When a `kvfs` filesystem is mounted, the free space bitmap is parsed and a linked list containing each free spot is allocated in RAM. This linked list uses the built-in FreeBSD `LIST_X(3)` macros. Then, whenever a new file must be created, an entry is simply popped off this list. Similarly, when a file is removed, the newly freed entry is added to the list.

```
/* represents a free key-value (inode, block) pair */
struct kvfs_freelist_entry {
    ino_t ino;          /* index of this inode on disk */
    LIST_ENTRY(kvfs_freelist_entry) entries;
};
```

Note that the free space list contains only the inode index of the key-value pair. The block number and location of free space bitmap entry can be calculated using the following macros:

```
/* convert ino to block index */
#define INO_TO_BLOCKNUM(ino) (((ino) / sizeof(struct kvfs_inode)) * BLOCKSIZE)

/* convert ino to free block byte offset */
#define INO_TO_FREE_BYTE(ino, mp) \
    (((ino) / sizeof(struct kvfs_inode) / 8) + mp->freelist_off)
```

```c
/* convert ino to bit mask where the free bit is set */
#define INO_TO_FREE_BIT_MASK(ino) (1 << ((ino) / sizeof(struct kvfs_inode) % 8))
```

2.1.3 Inode

The `kvfs` inode represents the state of each file on disk. Each inode is indexed by a unique value into the inode table on disk, and this index is also used as the block pointer for that key-value pair's data block. Inodes are allocated on disk in one contiguous segment.

```c
struct kvfs_inode {
    uint8_t key[20];    /* 160 bit key */
    uint16_t flags;     /* inode flags */
    uint16_t ref_count; /* reference count. currently always 1 */
    uint64_t timestamp; /* modification time in nanoseconds */
};
```

Each `kvfs_inode` is 256 bits in size, meaning 128 of them can fit in one 4KiB block.

Because each inode is allocated contiguously in a large table, `kvfs_inode` has a set of flags to help identify which entries are valid nodes. Currently only two flags are used:

```c
#define KVFS_INODE_ACTIVE 0x0001
#define KVFS_INODE_FREE 0x0002
```

2.1.3.1 In-memory

When a file in `kvfs` is created, a vnode representing this file is created, alongside a "helper" structure used to represent our inode in-memory. This structure contains the inode fields, alongside a pointer to the `struct kvfs_mount` associated with the filesystem, and extra metadata to make in-memory operations faster.

```c
struct kvfs_memnode {
    struct kvfs_mount *mp;  /* pointer to mount structure containing useful global information */
    struct vnode *vp;       /* pointer to associated vnode */
    ino_t ino;              /* index of kvfs_inode on disk */
    daddr_t lbn;            /* logical block number of "data" value on disk. */

    struct kvfs_inode inode; /* fields in kvfs inode */
};
```

Because the root inode does not exist on-disk, a special "invalid" `ino` is defined for the root. `kvfs` supports a maximum of 2^{30} inodes, each with a size of 32 bytes, so the root inode's index is set to an invalid offset which is greater than $2^{30} \times 32 = 2^{35}$. The value `0x800000008` was chosen arbitrarily.

2.2 Additional Data Structures and Algorithms

2.2.1 Mount Structure

The `kvfs_mount` structure is initialized when `kvfs` is first mounted. This structure is analogous to a `cdevsw` structure for a device driver, containing global data about the state of the filesytem, such as the underlying character device, the underlying disk device's vnode, and byte offsets of each segment on disk.

```c
struct kvfs_mount {
    struct mount *vfs;     /* vfs mount struct for this fs */
    struct g_consumer *cp; /* GEOM layer characver device consumer */
    struct vnode *devvp;   /* vnode for character device mounted */
    struct cdev *cdev;     /* character device mounted */

    off_t freelist_off;   /* data offset of freelist bitmap */
    off_t inode_off;      /* data offset of inode allocation table */
```

```
    off_t data_off;       /* data offset of data blocks */
    uint32_t block_count; /* number of data blocks in this filesystem. */
    uint64_t flags;

    LIST_HEAD(freelist_head, kvfs_freelist_entry) freelist_head;
    uint32_t freelist_count; /* number of free blocks */
};
```

2.3 Initializing the Filesystem – `mkkvfs`

The `mkkvfs` tool allows the user to format a disk device with the `kvfs` filesystem. This program will allocate space for each section as described in the Disk Layout section, and write these sections to disk.

In order to allocate each of the variable-size segments, an iterative algorithm is performed. The total size in bytes N of the disk is first found. We then guess how many blocks should fit in this space, subtracting the space used by the superblock, free space bitmap, and inode allocation. We must also be careful to insert padding after each section, to align it to the size of a disk block for performance reads and writes. We then iterate on this guess, checking to see if our guessed block count would match the target disk size.

Python-syntax pseudocode for this algorithm is presented below:

```
# initial guess
blocks = disksize / BLOCKSIZE / 2
# how much to increase or decrease our guess on each iteration
delta = disksize / 16;

while True:
    inode_count = blocks
    free_bitmap = ceil(blocks / 8)
    sum = PAD(sizeof(struct kvfs_superblock))
        + PAD(free_bitmap)
        + PAD(inode_count * sizeof(struct kvfs_inode))
        + blocks * BLOCKSIZE

    # check if we found the solution
    if disksize == sum:
        break

    # improve guess
    if sum > disksize:
        # reduce delta each time we overshoot target
        blocks -= delta
        delta /= 2
    else if sum < disksize:
        blocks += delta

    if delta < 1:
        error("could not converge solution")
```

Using the calculated sizes for each section, we figure out the offsets on-disk (making sure to include padding), and then write the superblock.

```
sb->freelist_off = PAD(sizeof(struct superblock));
sb->inode_off = PAD(free_bitmap) + sb->freelist_off;
sb->data_off = PAD(inode_count * sizeof(struct kvfs_inode)) + sb->inode_off;
```

Finally, we write out each of the sections to disk. When writing the inode table, we are careful to write each with the `KVFS_INODE_FREE` flag set.

2.4 VFS Operations

`kvfs` sits in the VFS layer, and as such, implements the VFS and vnode operations required for a filesystem of this type.

2.4.1 `VFS_MOUNT`

This is the entry point of the filesystem, and is the first routine called (just after `VFS_INIT`).

The structure of `VFS_MOUNT` follows this basic pesudocode:

- Look up mount point, verify that it is a disk device
  - Save the underlying disk's vnode
- Open a new GEOM layer character device for the disk
  - Increment reference count
- Allocate `struct kvfs_mount` and store useful pointers
- Set mount flags in `struct mount`
- Get new filesystem id
- Read and verify superblock
  - If superblock invalid, unwind and error
  - Store offsets from superblock in `struct kvfs_mount`
- Read freelist and allocate linked list
- Tell VFS we are finished mounting

2.4.2 `VFS_UNMOUNT`

Unmount finishes all pending I/O operations and frees any allocated resources.

- `vflush()` all allocated vnodes
- Close GEOM character device
- Unref cdev of disk
- Unref vnode of disk device
- Free any allocated structures (free list, `kvfs_mount`)
- Set mount flags to indicate that we are unmounted.

2.4.3 `VFS_INIT`

A `uma` zone is allocated to make allocating `kvfs_memnode` efficient.

2.4.4 `VFS_UNINIT`

The none `uma` zone is de-allocated.

2.4.5 `VFS_ROOT`

Root is called by the kernel's VFS driver code, directly after `VFS_MOUNT` during the mounting process. `VFS_ROOT` simply calls `VFS_VGET` with a special "root inode".

2.4.6 `VFS_VGET`

`VFS_VGET` can be used to "look up" an inode number (ino) and translate it to a vnode. `VFS_VGET` is used in the lookup routine, to allocate a new vnode/inode, or retrieve an already existing one.

- Retrieve cached vnode with `vfs_hash_get`.
  - If found, return it; otherwise continue
- Allocate a new vnode and lock it

- Associate the vnode with our filesystem mount with `insmntque`
- Insert new vnode into the vfs hash with `vfs_hash_insert`
- Allocate a `kvfs_memnode` for the vnode
- If ino != ROOT_INO, look up the inode on disk:
  - If inode does not exist, initialize it and write back
  - Otherwise, return new vnode
- Set vnode type and flags
- Return new vnode

The vnode operation `VFS_VGET` is a wrapper around an internal function that also allows the user to specify the filename or key for the new inode. This is so the new inode does not have to be written back in two separate steps.

2.4.7 `VFS_STATFS`

Statfs returns some basic information about the filesystem, such as the block size, the total number of blocks, the total number of files in use, etc. Almost all of this information is simply copied over from the `kvfs_mount` structure.

2.4.8 `VFS_SYNC`

Sync iterates over all vnodes that have been allocated, and runs `VOP_FSYNC` on each of them.

2.5 Vnode Operations

The default vnode operations are used whenever possible, such as `VOP_LOCK(), VOP_UNLOCK()`, etc. However, the majority of vnode operations must be implemented by `kvfs`.

Note that the following vnode handlers are completely unsupported, and are initialized with the special `VOP_EOPNOTSUPP` handler which simply returns an `EOPNOTSUPP` error to the caller:

- `VOP_MKDIR`
- `VOP_RMDIR`
- `VOP_LINK`
- `VOP_SYMLINK`
- `VOP_READLINK`
- `VOP_MKNOD`

2.5.1 `VOP_LOOKUP`

`VOP_LOOKUP` is arguably the most important vnode operation for a filesystem. The upper VFS layers will call our `VOP_LOOKUP` function before any operation on a file descriptor such as `read(2)`, `write(2)`, `unlink(2)`, etc.

Our lookup follows this basic structure:

- If name == `'.'`, they are looking up the root: increment refcount on the directory vnode and return
  - We don't need to implement lookup for `'..'`, since the upper layers do that for us when the lookup directory is filesystem root.
- Check that the passed filename is a valid 40-digit hexidecimal string. If not, return `EINVAL`.
  - This means trying to look up any file which is not valid will return `EINVAL`, not `ENOENT`.
- For each non-free inode on disk:
  - If name == inode.key, use VFS_VGET to find and return the vnode for this file.
  - Otherwise, keep checking
- At this point, file was not found, so return `ENOENT`.

2.5.2 VOP_CREATE

Create is used to create a new file. Because VOP_LOOKUP is called first, we assume the name is valid.

- Pop an entry from the free list
- Update free bitmap on disk to mark the new entry
- Allocate inode and vnode for new file using VFS_VGET

2.5.3 VOP_OPEN / VOP_CLOSE

Because we are not using the virtual memory backing (managed by vfs_vm_object), open and close are stubs which do nothing.

2.5.4 VOP_ACCESS

kvfs does not implement access control, so VOP_ACCESS always succeeds.

2.5.5 VOP_GETATTR

- If the vnode is the filesystem root, set type to VDIR and size to 0.
- If the vnode is a regular file, write out the vattr fields from the inode.
  - Deserialize the 64-bit nanosecond timestamp to a struct timespec.

2.5.6 VOP_SETATTR

- If the va_mtime field (modification time) is not set to VNOVAL indicating that it should be updated, we update the inode's time stamp in-memory, and write it to disk.

2.5.7 VOP_READ

- Verify that arguments are valid:
  - can't read a directory
  - can't read if uio offset is negative
- If remaining uio data is 0, or the offset is more than 1 block, return
- bread() the block containing the data value for this key
- while the requested amount is nonzero:
  - copy out as much data as we can to user using uiomove

2.5.8 VOP_WRITE

- Verify that arguments are valid:
  - can't read a directory
- If size of write is 0, return
- If user requested O_APPEND flag, error, since we can't write more than 1 block.
- Read the data block associated with the file
- While the requested write amount is nonzero:
  - copy in data to block buffer using uiomove
- write back buffer

2.5.9 VOP_FSYNC

Inspired by fsync in ext2fs.

- Use vop_stdfsync on this vnode
- If request did not ask to wait, try to fsync the device vnode. Make sure to lock it first.

2.5.10 `VOP_REMOVE`

Performed in "soft update" order:

- Zero out inode for this file
- Zero out data blocks for this file
- Add block to free list
- Update free list bitmap on disk

2.5.11 `VOP_RENAME`

- Check for cross-device rename, fail if target is outside our mount.
- If "destination" file exists, remove it
- Lock "from" vnode
- Overwrite name entry in "from" file with new name
- Update inode on disk
- Unlock "from" vnode

2.5.12 `VOP_READDIR`

- Write out entries for `.` and `..`
- For each non-free inode on disk:
  - Write out `struct dirent` for each entry

2.5.13 `VOP_INACTIVE`

- If file was deleted, we can recycle vnode with `vrecycle()`.

2.5.14 `VOP_STRATEGY`

Transform the logical block number in a `struct buf` to a physical block number, and call `BO_STRATEGY` to read or write from the buffer.

3 Testing

Basic functionality was tested with user-space tools like `cat`, `touch`, `rm`, `mv`, `stat`, and `ls`. Additionally, syscalls like `open(2)` were tested using a test driver, located in `tests/`

4 Limitations and Known Issues

- While storing inodes in a continuous table is very simple, a major consequence of this simplicity is that file lookups are always $O(n)$. Because we do not implement a directory hierarchy, a large `kvfs` partition will always have very slow lookups. Because `VOP_LOOKUP()` is called before pretty much any I/O operation, this means that I/O is overall quite slow.
  - This could be improved without changing the design by using a hash-map based lookup cache.
- `VOP_WRITE` does not appear to update the modification timestamp. However, `touch(1)` does.
- When using an editor like `vi(1)` or `nano(1)`, writing to a file will panic the kernel. This appears to be because the editor will attempt to write past the end of a block which does not belong to us.
- `VFS_SYNC` and `VOP_FSYNC` are currently broken. Instead, we always synchronously write the buffers with `bwrite()`
  - Because if this, `O_ASYNC` is not supported.
  - This means that `O_SYNC` is technically the "default mode" .

References

The following files in the FreeBSD source tree were consulted. Unless otherwise specified in the source, code from these files was not copied. Instead, the files were used as a reference and studied to understand the various VFS and vnode operations.

- `/usr/src/sys/fs/msdosfs/`
  - `msdosfsmount.h`
  - `msdosfs_vfsops.c`
  - `msdosfs_vnops.c`
  - `msdosfs_denode.c`
  - `msdosfs_lookup.c`
- `/usr/src/sys/fs/udf/`
  - `udf_vfsops.c`
  - `udf_vnops.c`
- `/usr/src/sys/fs/nullfs/`
  - `null_vnops.c`
  - `null_vfsops.c`
  - `null_subr.c`
- `/usr/src/sys/fs/cd9660/`
  - `cd9660_vnops.c`
  - `cd9660_vfsops.c`
- `/usr/src/sys/fs/autofs/`
  - `autofs_vfsops.c`
  - `autofs_vnops.c`
- `/usr/src/sys/fs/tmpfs/`
  - `tmpfs_vfsops.c`
  - `tmpfs_vnops.c`
- `/usr/src/sys/fs/ext2fs/`
  - `ext2_vnops.c`
  - `ext2_vfsops.c`
  - `ext2_lookup.c`
  - `ext2_alloc.c`
  - `ext2_inode.c`
  - `ext2_dinode.h`
- `/usr/src/sys/fs/devfs/`
  - `devfs_vnops.c`
  - `devfs_vfsops.c`
- `/usr/src/sys/fs/nfsserver/`
  - `nfs_nfsdsubs.c`
- `/usr/src/sys/ufs`
  - `ufs/ufs_vfsops.c`
  - `ufs/ufs_vnops.c`
  - `ffs/ffs_vfsops.c`
  - `ffs/ffs_vnops.c`
- `/usr/src/sys/kern/`
  - `vfs_bio.c`
  - `vfs_mount.c`
  - `vfs_lookup.c`
  - `vfs_subr.c`
- `/usr/src/sys/sys/`
  - `vnode.h`
  - `buf.h`

Additionally, the following web resources were used:

- https://man.netbsd.org/NetBSD-9.2-STABLE/buffercache.9
- https://netbsd.org/docs/internals/en/chap-file-system.html
- https://stackoverflow.com/a/45233496/71441

The filesystem layout diagram was created using https://diagrams.net.