CMPSCI 577 Operating Systems Design and Implementation **Spring 2020**

Lecture 18: Apr 02, 2020

Lecturer: ***Prashant Shenoy*** *Scribe:* ***Pavan Reddy Bommana***

18.1 Virtual file system(VFS)

VFS is a layer within the OS kernel that allows to us have different file systems present in the OS and provide an abstraction that allows access to these file systems. This sits above the implementation of different file systems within the kernel. Different file systems are underneath the VFS. System calls that are file system-specific always come to the VFS first, which then examines which file system is responsible for processing that call and forwards that call to the specific module that implements the file system. This mechanism helps us to easily add different file systems to the OS. In MINIX, VFS sits between the user processes and the filesystem processes that implement specific filesystem functionality. Each file system in MINIX is going to be a separate user process that is going to interact with the VFS.

18.2 VFS functionality

1) Its main function is to act as an abstraction layer to the file server processes.

2) It implements POSIX system call interface that allows user processes to invoke these system calls and also supports additional Lib C calls.

3) The two classes of system calls implemented in VFS are:

- **i)** File management system calls
- **ii)** Process management system calls that are tied closely to the file system.

4) Track endpoints for device drivers.

18.3 Control flow

The user process executes a read system call. VFS verifies that the read is on a valid file. It forwards the request to the concerned file server. File server reads the data using block drivers, device drivers, hard disk etc., Once the data is read, it is copied back to the user process and a reply is sent to the VFS indicating that the request has been executed. The VFS then replies to the user process that the operation is complete.

18.4 VFS Worker Threads

VFS process implements threads to handle concurrent IO requests. As MINIX has no support for kernel level threading, VFS threading is implemented using a user-level threading library called 'systhread'. VFS works through a standard master-slave architecture. Main thread waits for requests. There are a set of

worker threads. Whenever a main thread receives a request, it looks for an idle worker thread and hands that request to that worker thread. If all the worker threads are busy and VFS gets a new IO request, then the new requests will be queued. In more sophisticated servers, the size of worker thread pool is dynamic based on the load. In simple systems like MINIX, this size is static. VFS server uses these threads to interact with all the file server processes. VFS drives all file Servers and drivers asynchronously which means that while waiting for a reply, a worker thread is blocked and other workers can keep processing requests.

18.4.1 Types of worker threads

1) Normal threads - Handle requests from user processes.

2) System threads - Handle requests that come from other system tasks such as Process manager or kernel.

3) Deadlock threads - Handles deadlocks in the system.

18.5 VFS Data Structures

1) fproc: Data structure that holds file tables for each process.

2) vmnt: Data structure that tracks all the mounted file systems that are currently present in the system. It has information such as device major number, minor number, the mount flags etc.,

3) vnode: Data structure that tracks all the inodes in the system. Inodes of all the disk files are tracked in the vnode table.

4) File position table: For every open file, there is an offset that tells you where that file pointer is pointing to. This data structure is going to track that offset.

5) lock: Allows to lock certain parts of our file. It is used in advisory locking at the process level but not within the kernel.

6) select : Used to implement the select system call.

7) dmap: Used to map from the device major number to the specific device driver, so that you can forward the request to the appropriate device driver.

18.6 Locking in VFS

18.6.1 Locking requirements

1) Consistency of Replicated Data: Any system calls that update file sizes should be mutually exclusive from other system calls.

2) Isolation of System Calls: When system calls are composed of multiple requests, we need to ensure that result of a system call should not be impacted by result of another system call.

3) Integrity of objects: The locking solution is not perfect as lock acquiring process is a blocking operation, where an integrity of the locks itself requires extra locking, to break this loop we will heavily rely on the non-preemptiveness of the threading model to prevent this where possible.

4) No deadlock: Conflicts between locking of different types of objects can be avoided by keeping a locking order of object types. This ensures that there are no cycles, thus preventing a deadlock.

5) No starvation: The VFS must guarantee an execution time for all system calls.

6) The VFS must ensure that execution of one request is not blocking other requests. Also, no read-only operation on a regular file must block an independent read call to that file.

18.6.2 Types of locks

1) TLL_READ: Typical read lock that allows multiple threads to read.

2) TLL_WRITE: Write lock which allows one writer to hold the lock. No other reader or writer is allowed to hold a lock.

3) TLL_READSER: Allows to hold read locks on multiple parts of file rather than the complete file.

18.6.3 Locking order

In order to avoid deadlocks, VFS we uses the following order: fproc - $[exec] - vmnt - vnode - filp - [blockspecialfile] - [dmap]$

This means that no thread may try to acquire a higher level lock while still holding a lower level lock. This implies that no thread may lock an fproc object while holding a vmnt lock, and no thread may lock a vmnt object while holding an (associated) vnode, etc.

18.6.4 vmnt locking

This is used to acquire a lock while mounting and unmounting processes. For example, unmount operation should fail when an inode is still in use. There are three types of vmnt locks.

1) VMNT_READ: This lock allows read operations to proceed in a concurrent fashion using TLL_CONCUR locking scheme.

2) VMNT_WRITE: This locks uses TLL_SERIAL scheme. This means, we are going to wait for all the readers to finish and then we can acquire the lock and proceed to write.

3) VMNT_EXCL: This lock blocks everyone until the write is finished using TLL_EXCL locking scheme.

18.6.5 vnode locking

It uses similar locking as vmnt locking.

18.6.6 Filp locking

The filp object has two main variables:

i) *filp_count:* denotes number of processes that have opened that file for reading or writing.

ii) *filp_position:* denotes the offset within the file for each of the process that opened that file.

Filp objects use mutex locks. System calls that involve a file descriptor most often access both the filp that the file descriptor links to, and the vnode that that filp links to. The locking order imposes that vnodes be locked before filps. Whenever a filp is obtained based one of a process's own file descriptors, the corresponding vnode is locked with a certain requested vnode locking level first.

18.6.7 Lock locking

We need mutual exclusion for lock integrity. We assume our threading model is nonpreemptive, this means that all access to those structures is already fully atomic.

18.6.8 Select locking

We use one global mutex covering all of the select code.

18.6.9 dmap locking

All device drivers mapping functions are implemented in a non blocking fashion, this removes the need for locking.

18.7 Crash recovery

VFS plays an important role in crash recovery.

1) Block device crash: The file system process that is interfacing with the device driver will try first to restart the device driver, if it is not able to do so, then the VFS initiates restart 5 times.

2) Character device crash: The VFS process is going to try to recover or restart the trackers directly.

3) File server crash: If the File server crashes, VFS marks all associated file descriptors as invalid and cancels ongoing and pending requests to that File Server. Resources that were in use by the File Server are cleaned up.

18.8 MINIX filesystem layout

A MINIX 3 file system is a logical, self-contained entity with i-nodes, directories, and data blocks. It can be stored on any block device, such as a floppy disk or a hard disk partition. Boot block, super block and i-nodes are metadata blocks of the filesystem present in that order on the device. Remaining blocks on the file system are the data blocks where actual data is stored. The relative size of the various components in the layout may vary from device to device, depending on their sizes. But all the components are always present and in the same order.

18.8.1 Boot block

The boot block contains executable code, where the size of the block is always 1024 bytes. When the computer is turned on, the hardware reads the boot block from the boot device into memory and begins executing its code. The boot block code begins the process of loading the operating system itself. Once the system has been booted, the boot block is not used any more. Not every disk drive can be used as a boot device, but to keep the structure uniform, every block device has a block reserved for boot block code. At the worst, this strategy wastes one block. To prevent the hardware from trying to boot an unbootable device, a magic number is placed at a known location in the boot block.

18.8.2 Super block

It appears directly after the boot block. It is 1024 bytes in size always. The main function of the super block is to tell the file system how big the various pieces of the file system are. Given the block size and the number of inodes, it is easy to calculate the size of the inode bitmap and the number of blocks of inodes. Although super block is stored on disk, when you mount the filesystem, a copy of super block is also kept in memory so that the MFS(MINIX file system) can use it to access other information. Whenever super block is modified in memory, it has to be written back to the disk as well.

18.8.3 Bitmaps

Minix tracks free inodes and zones using bitmaps. When a file is deleted, its corresponding inodes is deleted by marking the coresponding bits to 0. Disk storage is allocated in terms of 2^n blocks called zones. A zone is a method to allows allocating as much blocks next to each other (on the same cylinder) to safe load time. To create a new file the disk is searched for the first free i-node and then to allocate data for the file, the list of blocks is scanned to find at least one free block.

18.8.4 I-node

Minix i-node tracks meta-data and data blocks of the file. When a file is opened, its i-node is located and brought into the inode table in memory, where it remains until the file is closed. I-nodes are of fixed size (64 bytes).

18.8.5 Block cache

Block cache is basically an LRU cache. To read a block, the cache is first searched for the block number hash. If the block is used, the block counter is increased to forbid the eviction. Modified blocks are kept in memory until its evicted from the cache or the sync system call is executed. The modified blocks or dirty blocks are going to be written into the disk at a later point of time following write-back mechanism.