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

Lecture 22: Apr 16, 2020

Lecturer: ***Ahmed Ali-Eldin*** *Scribe:* ***Pavan Reddy Bommana***

22.1 Block Devices

Block devices are hardware devices distinguished by the random access of fixed-size blocks of data i.e., not necessary to access data in sequence. In Linux, all these devices will be mounted to the file system.
Ex: Hard disk, Bluray, Flash memory.
Block devices are much harder to manage compared to character devices. Character devices have very few possible I/O states compared to block devices. Hence, kernel has no subsystem for managing character devices. Character devices management is mostly done by the device drivers.
Block device management in Linux is done by a subsystem. The main goals for this subsystem are to increase the overall throughput, decrease the latency, have safety against failures and have fairness.

22.1.1 Anatomy of Block Device

Smallest addressable unit on a block device is called a sector. Sector sizes varies between devices and architectures, but are typically between 512 Bytes to 4KBs. These sectors are also called as device blocks or hard sectors. The software imposes its own smallest logically addressable unit, which is the block. The kernel performs all disk opertions in terms of blocks. The block size can be no smaller than the sector and must be a multiple of a sector. Kernel also requires that a block be no larger than the page size. Blocks are sometimes referred to as "filesystem blocks" or "I/O blocks".

22.1.2 Buffers and Buffer Heads

A block read by kernel, is stored in memory, in what we call as a buffer. The buffer serves as the object that represents a disk block in memory. A single page can hold one or more blocks in memory. The kernel requires some associated control information to accompany the data (such as from which block device and which specific block the buffer is). Each buffer is associated with a buffer head. The purpose of this buffer head is to describe the mapping between the on-disk block and the physical in-memory buffer. This function of acting as a descriptor for buffer-to-block mapping is the only role of buffer head in the kernel.

22.1.3 Bio layer

All block devices in Linux are represented by a generic disk structure called gendisk. It is supposed to encompass a generic view of all the different block devices. The bio layer carries read and write requests, and assorted other control requests, from the block_device, past the gendisk, and on to the driver. It basically acts as an intermediate layer the gendisk and the device driver. A bio identifies a target device, an offset in the linear address space of the device, a request (typically READ or WRITE), a size, and some memory where data will be copied to or from.

22.1.3.1 Bio Structure

Each block I/O request is represented by a bio structure struct bio data structure. This data structure is the basic container for block I/O operations. This structure represents block I/O operations that are active as a list of segments. A segment is a chunk of a buffer that is contiguous in memory. The bio structure provides the capability for the kernel to perform block I/O operations of even a single buffer from multiple locations in memory.

22.1.3.2 Important fields in the struct bio

1) Bi_io_vec: An array of bio_vec structures. These structures are used as lists of individual segments in a specific block I/O operation.

2) Bio_vec: Each bio_vec is treated as a vector of the form <page, offset, len>. It describes a specific segment, the physical page on which it lies, the location of the block as an offset into the page, the length of the block starting from the given offset.

22.1.4 Request Queues

Block devices maintain request queues to store their pending block I/O requests. Each block device request is represented by a request descriptor. The request queue contains a doubly linked list of requests and associated control information. As long as the request queue is nonempty, the block device driver associated with the queue grabs the request from the head of the queue and submits it to its associated block device.

22.1.4.1 Single Request Queue

For each block device, there is one single queue from which the device needs to process the different tasks. Traditionally, most storage devices were made up of a set of spinning circular platters with magnetic coating and a single head. Such a device can only process a single request at a time, and has a substantial cost in moving from one location on the platters to another. There is a scheduler called single-queue scheduler that schedules which request can process first from this single request queue. The three key tasks for this scheduler are :

(i) To collect multiple bios representing contiguous operations into a smaller number of requests that are large enough to make best use of the hardware but not so large that they exceed any limitations of the device.

(ii) To queue these requests in an order that minimizes seek time while not delaying important requests unduly. Relying on heuristics, aim to provide an optimal solution to this problem is the source of all the complexity.

(iii) To make these requests available to the underlying driver so it can pluck them off the queue when it is ready and to provide a mechanism for notification when those requests are complete.

22.1.4.2 Multiple Request Queue

Many devices today can accept multiple I/O requests at once. Also, as we get more and more processing cores in our systems, the locking overhead required to place requests from all cores into a single queue increases. These reasons justify the usage of multiple request queues.

Linux introduced software staging queues. Requests are added to these queues, controlled by a spinlock that should mostly be uncontended.

22.2 I/O Schedulers

I/O scheduler is between the page cache and disk. It schedules a queue of pending requests (I/O request queues). The requests are present in form of a tuple of (block #, read/write, buffer addr). The reordering of these requests based on some heuristic is called I/O scheduling. The I/O scheduler divides the resource of disk I/O among the pending block I/O requests in the system by merging and sorting of pending requests in the request queue.

22.2.0.1 Elevator Algorithm

It is developed by Linus. It follows the following steps:

(i) When a request is added to the queue, it is first checked against every other pending request to see whether it is a possible candidate for merging.

(ii) If a suitable location sector-wise is in the queue, the new request is inserted there. This keeps the queue sorted by physical location on disk.

(iii) Finally, if no such suitable insertion point exists, the request is inserted at the tail of the queue.

22.2.1 The NOOP Scheduler

This is the simplest available scheduler. It is obselete now. It provides minimal sorting of requests, never allowing a read to be moved ahead of a write or vice-versa. It allows one request to overtake another if, it is in line with the elevator algorithm. Apart from this simple sorting, this scheduler provides first-in-first-out queuing.

One issue with NOOP scheduler and elevator algorithm is that, they look at only spatial characteristics when it comes to the block I/O requests. This means that some requests that are quite far away in terms of sectors, can end up starved or severely delayed.

22.2.2 The Deadline Scheduler

The deadline Scheduler is designed to fix the above drawback of NOOP scheduler. This scheduler collects batches of either read or write requests that were all submitted at close to the same time. This scheduler maintains a request queue sorted by physical location on disk that is used for scheduling as long as a certain time quota did not expire. Once filled, this queue is used to schedule operations. In addition, read requests are sorted into a special read FIFO queue, and write requests are inserted into a special write FIFO queue.

22.2.3 The BFQ I/O Scheduler

The Budget Fair Queueing (BFQ) I/O Scheduler is a proportional-share (measured in number of sectors) storage-I/O schedule where each process/thread to be assigned a fraction of the I/O throughput. The algorithm explicitly privileges the I/O of time-sensitive applications(interactive and soft real-time applications).

BFQ algorithm has a mechanism to detect the requests from interactive and soft real-time applications within the queue and service them before anything else.

22.2.4 The Kyber I/O scheduler

This scheduler is intended for fast multiqueue devices. I/O requests passing through the Kyber scheduler are split into two primary queues, one for synchronous requests (reads) and one for asynchronous requests (writes). The idea is that the reads are much less tolerant to delays than writes because most applications will block on reads and wait but do not do so on writes. The number of operations (both reads and writes) sent to the dispatch queues (the queues that feed operations directly to the device) is strictly limited. This is to make sure that we don't end up with an endless number of reads starving all the writes and vice-versa.

22.3 The Linux Virtual Filesystem (VFS)

The VFS is a substantial piece of code, not just an API wrapper. It provides a level of abstraction between the user and the different file systems. It has many functionalities such as -

(1) Caches file system metadata (e.g., file names, attributes) .

(2) Coordinates data caching with the page cache.

(3) Enforces a common access control model .

(4) Implements complex, common routines, such as path lookup, file opening, and file handle management.

22.3.1 Primary Object Types of VFS

1. The superblock object, which represents a specific mounted filesystem.
2. The inode object, which represents a specific file.
3. The dentry object, which represents a directory entry, which is a single component of a path.
4. The file object, which represents an open file as associated with a process.

22.3.2 Special Filesystems

The special filesystems enable an administrator to manipulate some of the kernel data structures and to implement special features of the operating system. For example, the bdev filesystem for managing block device drivers, the pipefs special filesystem for Pipes, the proc filesystem mounted on proc, which acts as a General access point to kernel data structures.

22.3.3 Pipefs

Pipes are an interprocess communication mechanism present in UNIX systems since the beginning. A pipe is a one-way flow of data between processes. All data written by a process to the pipe is routed by the kernel to another process. Pipe provides a pair of file descriptors, one for read, one for write. In Linux, the support for pipe built on pipefs. Pipefs is kernel mounted file system and has no mount point. A pipe is implemented as a set of VFS objects, which have no corresponding disk images.

22.3.3.1 Pipes : More details

Pipes may be considered open files that have no corresponding image in the mounted filesystems. A process creates a new pipe by means of the pipe() system call, which returns a pair of file descriptors. The process may then pass these descriptors to its descendants through fork() thus sharing the pipe with them. The processes can read from the pipe by using the read() system call with the first file descriptor and write to the pipe by using write() system call with the second file descriptor.

22.3.4 Linux FS: Ext3 filesystem

Ext3 filesystem supports immutable files (files that cannot be modified, deleted, or renamed) and append-only files. The filesystem pre-allocates disk data blocks to regular files before they are actually used, thereby reducing file fragmentation. The filesystem partitions disk blocks into groups. Each group includes data blocks and inodes stored in adjacent tracks. As a result, files stored in a single block group can be accessed with a lower average disk seek time. Ext3 is a journalling filesystem and journaling avoids the time-consuming check that is automatically performed on a filesystem when it is abruptly unmounted.

22.4 Signals

Signals are basically opposite of an interrupt. A signal is a very short message that may be sent to a process or a group of processes. The only information given to the process is usually a number identifying the signal. There are a number of pre-identified signals that Linux has. These standard signals have no arguments, message, or other accompanying information. A set of macros whose names start with the prefix SIG is used to identify signals.

Signals serve two main purposes. They make a process aware that a specific event has occurred. They force a process to execute a signal handler function included in its code. Process can change how it handle certain signal. It can use a different signal-handler function, use a kernel-default signal handler, ignore the signal or block the signal.

22.4.1 Features

Signals may be sent at any time to a process whose state is usually unpredictable. Signals sent to a process that is not currently executing must be saved by the kernel until that process resumes execution. The two phases related to signal transmission are:

Signal Generation: The kernel updates a data structure of the destination process to represent that a new signal has been sent.

Signal Delivery: The kernel forces the destination process to react to the signal by changing its execution state, by starting the execution of a specified signal handler, or both. Once a signal has been delivered, all process descriptor info referring to its previous existence is canceled.

Kernel must remember which signals are blocked by each process. When switching from Kernel Mode to User Mode, kernel must check whether a signal for any process has arrived. Kernel must determine whether the signal can be ignored. It must handle the signal, which may require switching the process to a handler function at any point during its execution and restoring the original execution context after the function returns.