Operating Systems 2

The Block I/O Layer

Arkadiusz Chrobot

Department of Information Systems

June 4, 2025

Outline

Introduction

Block I/O devices require a more complex handling than character devices. There are several reasons for that. Block devices offer random data access, meaning that it is possible to directly specify a location on the medium from which data should be read or written. This implies that there is a way of changing the location of the block device data pointer in both directions: forward or backward. Almost all block I/O devices are equipped with a file system. The still most frequently used block devices are hard disks, but there are many more devices of that type (CDs, DVDs, other optical disks, Solid State Devices and other flash memory devices). The access time to these devices (particularly the hard disk) is one of the most important factors that have an impact on the overall performance of the computer system. That's why the Linux kernel programmers decided to implement a whole new subsystem for handling those devices, which is called *The Block I/O Layer*.

Buffers

Although the block I/O layer supports all block devices it has been designed primarily for hard disks. The block devices store data in units called sectors. The size of a single sector is usually 512B (there are several exceptions, like CDs). Most of the I/O operations involve more than one sector. That's why modern operating systems tend to use *blocks* instead of sectors. A single block is a sector or a group of adjacent sectors. In the Linux kernel the block size is smaller or equal to the size of a page[1]. Each block that is involved in an I/O operation has its *buffer* in the RAM and each buffer has a *buffer header*. The header stores all the information required for managing the buffer. Its data type is defined by the `struct buffer_head` structure. Among the data stored in the header is the state of the buffer kept in the `b_state` field. This information is described by one or several elements of the `bh_state_bits` enumeration.

[1]This limitation has been removed since the release of the 6.3 version of the kernel.

Buffers

Elements of `bh_state_bits`

The `BH_Uptodate` element means that the data in the block and in its buffer are the same. The `BH_Dirty` element indicates that the data in the buffer have been modified, but not yet written to the block in the device. The `BH_Lock` element denotes that the buffer is protected against concurrent access, because it takes a part in an ongoing I/O operation. The `BH_Req` element means that the buffer is used in an ongoing I/O request. The `BH_Update_Lock` element marks the first buffer from the group of buffers located on the same page that are protected against concurrent access, because they take a part in an ongoing I/O operation. The `BH_Mapped` indicates that the buffer is associated with a block in the device — the Linux kernel makes an overprovision of buffers, so not all of them are immediately associated with blocks. The `BH_New` element means that the buffer has been associated with a block, but it hasn't been used yet. The `BH_Async_Read` element denotes that the buffer is used in an asynchronous read operation.

Buffers

Elements of `bh_state_bits`

The `BH_Async_Write` element means that the buffer takes a part in an asynchronous write operation. The `BH_Delay` element indicates that the buffer has not been associated yet with a block in the device. The `BH_Boundary` element denotes a buffer belonging to a block that is the boundary of a group of blocks that form a continuous area on the medium, like for example a disk track. The `BH_Write_EIO` indicates that there was an error while the content of the buffer was stored on the medium. The `BH_Unwritten` element means that the buffer is associated with a block, but its data haven't been stored in that block yet. The `BH_Quiet` denotes that I/O errors associated with the buffer won't be reported. The `BH_Meta` element means that the buffer contains metadata. The `BH_Prio` element indicates that the buffer takes a part in a high-priority I/O operation. The `BH_Defer_Completion` element means that the buffer is involved in an I/O operation whose completion is deferred with the use of a work queue — it is an asynchronous operation.

Buffers

The `bh_state_bits` enumeration has one more element which informs that the rest of the most significant bits in the `b_state` field can be used by the device driver for its own use. The element is called `BH_PrivateStart`. One of the other fields of the header is the `b_count` field that is a reference counter. Its value is incremented by the `bh_get()` function and decremented with the use of the `bh_put()` function. Both of them are *inline* functions. The reference counter should be incremented before any operation on the buffer is performed. This prevents a premature deallocation of the buffer. The `b_dev` file contains an address of a structure that describes the block device storing the block associated with the buffer. The `b_blocknr` stores the number of the block. The page that contains the buffer is specified by the `b_page` field. The address within that page, from which the buffer area starts is stored in the `b_data` field and the size of the buffer is kept in the `b_size` field. There are other members of the header, but they are less interesting and won't be described here.

The Block Input-Output Structure

Buffer headers used to take a part in I/O operations in the kernel versions that predate the release of the 2.6 series. This caused serious efficiency issues, because a single read or write operation required using a lot of such headers scattered across the whole RAM. Moreover, the size of the header was almost the same as the size of the buffer. The kernel developers decided to remove some of the header fields and create another structure, called *BIO*, which represents an ongoing I/O operation with the use of a list of *segments*. The word “segment” in this context means a continuous part of a buffer. Buffers whose segments are the elements of the list don’t have to form a continuous area in the RAM. Moreover, the buffer can simultaneously take a part in several I/O operations thanks to the BIO structures. The most important members of the BIO structure are: `bi_io_vec`, `bi_vcnt` and `bi_iter`. The last one is a structure itself, that contains the `bi_idx` field. The first of them stores an address of an array of `bio_vec` structures, which is an implementation of the segments list.

The Block Input-Output Structure

Each element of the array is a structure with three fields: `bv_page`, `bv_offset` and `bv_len`. The first specifies the page where the segment is located, the second the offset in the page from where the segment starts, and the third the size of the segment. The `bi_io_vec` array describes the entire memory space consisting of the segments of buffers assigned to an I/O operation. The `bi_vcnt` field specifies how many elements of that array actually takes a part in the I/O operation. The currently processed element of the array is specified by the `bi_iter` field, whose value is constantly updated. Using this field allows the kernel to clone the BIO structure, which is beneficial for device drivers of such devices as RAIDs, because the kernel can set a different value of the `bi_idx` field for each of the BIO structure copies. This makes it possible to perform the I/O operation described by this structure in parallel. The BIO structure has its own reference counter which is incremented with the use of the `bio_get()` function and decremented with the help of `bio_put()` function.

The Block Input-Output Structure

The **`bi_private`** field of the BIO structure may store data belonging to the structure's creator. Using the BIO structure in the kernel has the following benefits:

- because the BIO structure uses the **`struct page`** structures, block I/O operations can use the high memory,
- the BIO structure can represent the regular I/O operations as well as direct I/O operations that do not use buffers,
- it is easier to perform an I/O operation whose data come from many pages scattered across the RAM (so-called scatter-gather or vectored block I/O operations),
- managing the BIO structure is easier than managing the buffer header.

I/O Schedulers

Most of the device drivers maintain a queue of I/O requests for the device they are handling. These queues are called *request queues* and are represented by the `request_queue` structure which stores control data needed for managing the queue and a pointer to a doubly linked list of requests. Each request in the queue is represented by the `struct request` structure. If the queue is not empty than the driver takes the first request from the queue and performs it. Each request can contain many BIO structures that represents a specific I/O operation with the use of the segments. The relation between the two latter structures is shown in the Figure 1.

I/O Schedulers

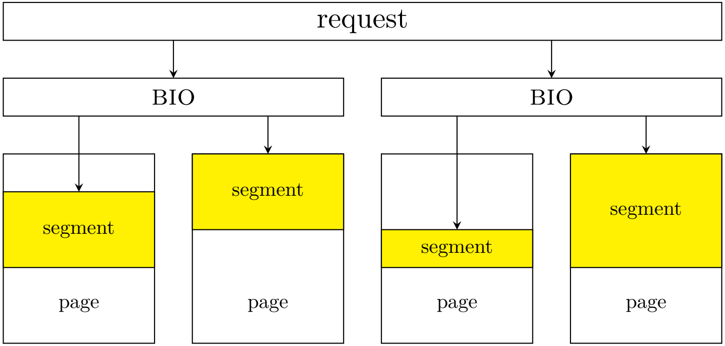

Figure 1: The relations between data structures used by the Block I/O Layer

I/O Schedulers

For scheduling the requests in the queue is responsible the I/O scheduler whose job is to minimize the data access time[2]. It allows a better average bandwidth utilization and prevents request starvation to appear. Basically the I/O scheduler performs two operations on I/O requests: merging and sorting[3].

[2]For example by reducing movements of heads in such block devices as a hard disk.

[3]Not to be confused with the merge sort algorithm.

I/O Schedulers

When a new request is created the I/O scheduler tries to merge it with requests that are already in the queue and relate to adjacent blocks. If the scheduler fails to do that then it tries to add the new request among other requests in the queue that are associated with blocks located in its proximity. These operations reduce the need for frequent changing of the block device data pointer movement direction. This behaviour of the I/O scheduler is defined by the LOOK algorithm described in many operating system textbooks. The Linux kernel offers its users a choice of at least three I/O scheduling algorithms[4]. Before the 2.6 kernel series was released there was only one I/O scheduler algorithm called the *Linus Elevator*[5]. This algorithm uses a *front* and *back* merging which means that the new request can be merged at the start or at the end of a cluster of requests that relate to the adjacent blocks in the device.

[4]The number sometimes changes with the release of a new kernel version.

[5]I/O scheduling algorithms are often called *elevators*.

I/O Schedulers

The Linus Elevator

The back merging happens more often than the front merging. If the new request cannot be merged with others then the I/O scheduler switches to sorting i.e. it tries to add the request among other requests that relate to blocks located in its proximity. If it fails to do that then it adds the new request at the end of the request queue. The scheduler does it also when it finds a request which is about to expire. This should prevent starvation of the request, but on the contrary it sometimes causes this issue.

I/O Schedulers

Deadline I/O Scheduler

In the 2.6 series of Linux kernels the Linus Elevator I/O scheduler has been replaced with three other algorithms. The first of them is the *Deadline I/O Scheduler* which prevents request starvation and gives precedence to the read requests over the write requests. The read delays have more impact on user-space application performance than write delays. The Deadline I/O Scheduler maintains three queues: the *sorted queue*, the *read FIFO queue* and the *write FIFO queue*. When a new request is created it is added to the sorted queue where the sorting and merging happens, just like in the Linus Elevator. Simultaneously it is also added to the write FIFO or the read FIFO depending on what type of request it is. The Deadline I/O Scheduler assigns a 500 milliseconds deadline to each read request and 5 second deadline to the write request. Normally the request from the front of the sorted queue is removed and added to the dispatch queue — the queue managed by the device driver.

I/O Schedulers

Anticipatory I/O Scheduler

However, when one of the request from the FIFO queues is close to expiring then this request is added to the dispatch queue. Another scheduler in the 2.6 series *was* the *Anticipatory I/O Scheduler*. It operated similarly to the Deadline I/O scheduler but it tried to avoid interrupting a stream of write requests by a single read requests. If it detected such a request it would stop handling other request for 6 ms — this time could be configured. If during the time another read request occurred then the Anticipatory I/O Scheduler would handle it immediately. This behaviour was beneficial if such cases happened a lot. Otherwise the waiting time could be wasted. To prevent such an issue the Anticipatory I/O Scheduler gathered statistics of the user-space processes I/O operations and used heuristic functions to predict if the new read operation would be followed by a next one. The Anticipatory I/O Scheduler was the default I/O scheduler in the 2.6 series until the release of the 2.6.18 kernel version. In the last version of the series 2.6.23 it was entirely removed from the kernel.

I/O Schedulers

The CFQ I/O Scheduler

The *Completely Fair Queuing I/O Scheduler* has been introduced in the kernel in the 2.6.6 version and it became the default scheduler in the 2.6.18 version (several distributions used it earlier as a default I/O scheduler). Its behaviour can be shortly described as a mixture of the multi-level queuing, the round-robin algorithm and the anticipatory I/O scheduling. The CFQ I/O Scheduler introduces a new property of user-space processes, the *I/O priority*. This scheduler also allocates for each of these processes a queue, implemented as a red-black tree, for synchronous I/O operations[6]. It also maintains several queues for the asynchronous I/O operations, which are shared by all user-space processes. The CFQ I/O Scheduler services the queues in a round-robin fashion starting from the queue of the process with the highest I/O priority. From each of the queues it takes as many I/O requests as the time slice assigned to the queue allows it.

[6]Synchronous I/O operations require the process to wait for their completion.

I/O Schedulers

The Noop Scheduler

The time slice is also specified by the I/O priority. However, if the CFQ I/O Scheduler empties the queue before the time slice expires, the scheduler can use the remaining time to wait for new I/O requests to occur in the queue. If that happens the requests are handled immediately. After the CFQ I/O Scheduler services all the queues associated with processes it starts handling the queues for the asynchronous I/O operations, although in their case it doesn't apply the waiting. Because the Anticipatory I/O Scheduler in some respect copies the behaviour of the CFQ I/O Scheduler, but its efficiency is worse, it has been removed from the kernel and replaced by the latter scheduler.

The last I/O scheduler is the *Noop I/O Scheduler*[7]. This scheduler performs only the merging operation on request queue and it is used with devices that offer a truly random access to data, like the flash memory storage devices.

[7] The name is derived from the "no-operations" word.

I/O Schedulers

The default I/O scheduler can be changed when the kernel is configured for compiling or even during kernel's runtime. The second option requires only modifying of one of the files in the `/sys` directory, for example the `/sys/block/sda/queue/scheduler` file. The following command displays the content of this file:

```
cat /sys/block/sda/queue/scheduler
```

If the result is like this:

```
noop deadline [cfq]
```

then it means that the CFQ I/O Scheduler is the default I/O scheduler. To change it to the Deadline I/O Scheduler the `root` user can use the following command:

```
echo deadline > /sys/block/sda/queue/scheduler
```

The New Block I/O Layer

A major rework of the Block I/O Layer took place in the 3.13 release of the Linux kernel. At that time the *Solid State Drives* (SSDs) become more common. These devices offer a far more better performance than hard disks for which the original Block I/O Layer was designed (millions of operations per second vs. hundreds of operations per second.). The Block I/O Layer became a bottleneck for SSDs, especially in the multiprocessor computers. The Linux kernel programmers decided to add third mode of operation for this layer. The first mode is for block devices that require no request queue, the second is for devices that require a single request queue, and the third is for SSDs. This mode of operation of the Block I/O Layer is so different from the previous two, that the kernel developers started to call it the *New Block I/O Layer*. In this mode each CPU (or a node in the NUMA architecture-based computer system) has its own *software request queue* which *isn't* protected with a spin lock. The only operation that originally was performed on this queue was merging of adjacent I/O requests.

The New Block I/O Layer

Each SSD is equipped with at least one, but usually several *hardware request queues*. The number of these queues is determined by the device driver when it is initialized and it depends on the device capability of handling the I/O requests in parallel. Requests from the software queues are moved to the hardware queues and then are serviced by the SSD. The new mode of the Block I/O Layer, called a multiqueue mode, replaced the single queue mode in 5.3 and newer versions of the Linux kernel.

The New Block I/O Layer
The Kyber I/O Scheduler

Initially, the kernel programmers thought that no scheduling is required for the software request queues, but it proved to be helpful in improving the efficiency of the slower SSDs and servicing the priorities of I/O requests coming from various user-space processes. In the 4.11 kernel version, the Deadline I/O Scheduler has been modified to service these queues. In the 4.12 version two I/O schedulers designed for this purpose have been added. The first of them — the *Kyber I/O Scheduler* — is much simpler than the other. Its goal is to reduce the latency of I/O requests. To this end it splits each software request queue into two, one for the synchronous I/O operations and the other for the asynchronous I/O operations. The deadline for the first type of I/O operations is 2 ms and 10 ms for the second type. The Kyber I/O Scheduler moves the I/O requests from the software request queues to the hardware request queues in such a way that the latter are as short as possible. This assures a short time of processing requests.

The New Block I/O Layer

The Kyber I/O Scheduler

The maximum number of I/O requests in a hardware request queue is determined by the time of processing previous I/O requests.

The New Block I/O Layer

The BFQ I/O Scheduler

The *Budget Fair Queuing (BFQ) I/O Scheduler* was planned for the single queue mode of operation, but eventually has been redesigned for the multiqueue mode. It is modelled after the CFQ I/O Scheduler, but it also has some features of the CFS process scheduler. The BFQ I/O Scheduler assigns to each of processes a number of sectors (the *budget*) that it is allowed to transmit in the current round of I/O operations scheduling. The input data for calculating the budget are the *I/O weight* of the process and its behaviour in the previous rounds of I/O scheduling. The calculations are quite complex, but the resulting budget must not exceed the global limit. The process's budget is its share in the block I/O device bandwidth, which is determined with the use of heuristics. The I/O requests of processes with a lower budget are handled before the I/O requests of processes with a larger budget. Each process also has a time slice when it has to use its budget.

The New Block I/O Layer

The BFQ I/O Scheduler

If a process manages to use all its budget before the time slice expires and its last I/O operation is a synchronous one, then the BFQ I/O Scheduler waits for a new request from this process, just like the Anticipatory and CFQ I/O Schedulers do. Several complicated heuristics are applied to improve the performance of the BFQ I/O Scheduler. Their detailed description as well as the description of the BFQ I/O Scheduler itself can be found in an article entitled "The BFQ I/O scheduler" by Jonathan Corbet, available here: https://lwn.net/Articles/601799/.

Questions

?

The End

Thank You for Your attention!