14

The Block I/O Layer

Block devices are hardware devices distinguished by the random (that is, not necessarily sequential) access of fixed-size chunks of data. The fixed-size chunks of data are called *blocks*. The most common block device is a hard disk, but many other block devices exist, such as floppy drives, Blu-ray readers, and flash memory. Notice how these are all devices on which you mount a filesystem—filesystems are the lingua franca of block devices.

The other basic type of device is a *character device*. Character devices, or *char* devices, are accessed as a stream of sequential data, one byte after another. Example character devices are serial ports and keyboards. If the hardware device is accessed as a stream of data, it is implemented as a character device. On the other hand, if the device is accessed randomly (nonsequentially), it is a block device.

The difference comes down to whether the device accesses data randomly—in other words, whether the device can *seek* to one position from another. As an example, consider the keyboard. As a driver, the keyboard provides a stream of data. If you type *wolf*, the keyboard driver returns a stream with those four letters in exactly that order. Reading the letters out of order, or reading any letter but the next one in the stream, makes little sense. The keyboard driver is thus a char device; the device provides a stream of characters that the user types onto the keyboard. Reading from the keyboard returns a stream first with *w*, then *o*, then *l*, and ultimately *f*. When no keystrokes are waiting, the stream is empty. A hard drive, conversely, is quite different. The hard drive's driver might ask to read the contents of one arbitrary block and then read the contents of a different block; the blocks need not be consecutive. The hard disk's data is accessed randomly, and not as a stream; therefore, the hard disk is a block device.

Managing block devices in the kernel requires more care, preparation, and work than managing character devices. Character devices have only one position—the current one—whereas block devices must be able to navigate back and forth between any location on the media. Indeed, the kernel does not have to provide an entire subsystem dedicated to the management of character devices, but block devices receive exactly that. Such a subsystem is a necessity partly because of the complexity of block devices. A large reason, however, for such extensive support is that block devices are quite performance

sensitive; getting every last drop out of your hard disk is much more important than squeezing an extra percent of speed out of your keyboard. Furthermore, as you will see, the complexity of block devices provides a lot of room for such optimizations. The topic of this chapter is how the kernel manages block devices and their requests. This part of the kernel is known as the *block I/O layer*. Interestingly, revamping the block I/O layer was the primary goal for the 2.5 development kernel. This chapter covers the all-new block I/O layer in the 2.6 kernel.

Anatomy of a Block Device

The smallest addressable unit on a block device is a *sector*. Sectors come in various powers of two, but 512 bytes is the most common size. The sector size is a physical property of the device, and the sector is the fundamental unit of all block devices—the device cannot address or operate on a unit smaller than the sector, although many block devices can operate on multiple sectors at one time. Most block devices have 512-byte sectors, although other sizes are common. For example, many CD-ROM discs have 2-kilobyte sectors.

Software has different goals and therefore imposes its own smallest logically addressable unit, which is the *block*. The block is an abstraction of the filesystem—filesystems can be accessed only in multiples of a block. Although the physical device is addressable at the sector level, the kernel performs all disk operations in terms of blocks. Because the device's smallest addressable unit is the sector, the block size can be no smaller than the sector and must be a multiple of a sector. Furthermore, the kernel (as with hardware and the sector) needs the block to be a power of two. The kernel also requires that a block be no larger than the page size (see Chapter 12, "Memory Management," and Chapter 19, "Portability").[1] Therefore, block sizes are a power-of-two multiple of the sector size and are not greater than the page size. Common block sizes are 512 bytes, 1 kilobyte, and 4 kilobytes.

Somewhat confusingly, some people refer to sectors and blocks with different names. Sectors, the smallest addressable unit to the device, are sometimes called "hard sectors" or "device blocks." Meanwhile, blocks, the smallest addressable unit to the filesystem, are sometimes referred to as "filesystem blocks" or "I/O blocks." This chapter continues to call the two notions *sectors* and *blocks*, but you should keep these other terms in mind. Figure 14.1 is a diagram of the relationship between sectors and buffers.

Other terminology, at least with respect to hard disks, is common—terms such as *clusters*, *cylinders*, and *heads*. Those notions are specific only to certain block devices and, for the most part, are invisible to user-space software. The reason that the sector is important

[1] *This is an artificial constraint that could go away in the future. Forcing the block to remain equal to or smaller than the page size, however, simplifies the kernel.*

to the kernel is because all device I/O must be done in units of sectors. In turn, the higher-level concept used by the kernel—blocks—is built on top of sectors.

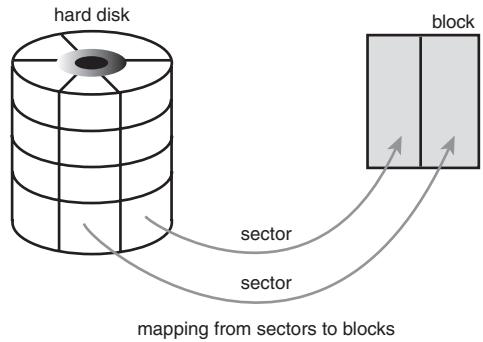

Figure 14.1 Relationship between sectors and blocks.

Buffers and Buffer Heads

When a block is stored in memory—say, after a read or pending a write—it is stored in a *buffer*. Each buffer is associated with exactly one block. The buffer serves as the object that represents a disk block in memory. Recall that a block is composed of one or more sectors but is no more than a page in size. Therefore, a single page can hold one or more blocks in memory. Because 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 descriptor. The descriptor is called a *buffer head* and is of type struct `buffer_head`. The `buffer_head` structure holds all the information that the kernel needs to manipulate buffers and is defined in `<linux/buffer_head.h>`.

Take a look at this structure, with comments describing each field:

```
struct buffer_head {
        unsigned long b_state;       /* buffer state flags */
        struct buffer_head *b_this_page; /* list of page's buffers */
        struct page *b_page;         /* associated page */
        sector_t b_blocknr;          /* starting block number */
        size_t b_size;               /* size of mapping */
        char *b_data;                /* pointer to data within the page */
        struct block_device *b_bdev; /* associated block device */
        bh_end_io_t *b_end_io;       /* I/O completion */
        void *b_private;             /* reserved for b_end_io */
        struct list_head b_assoc_buffers;  /* associated mappings */
        struct address_space *b_assoc_map; /* associated address space */
        atomic_t b_count;            /* use count */
};
```

The b_state field specifies the state of this particular buffer. It can be one or more of the flags in Table 14.1. The legal flags are stored in the bh_state_bits enumeration, which is defined in <linux/buffer_head.h>.

Table 14.1 **bh_state Flags**

Status Flag	Meaning
BH_Uptodate	Buffer contains valid data.
BH_Dirty	Buffer is dirty. (The contents of the buffer are newer than the contents of the block on disk and therefore the buffer must eventually be written back to disk.)
BH_Lock	Buffer is undergoing disk I/O and is locked to prevent concurrent access.
BH_Req	Buffer is involved in an I/O request.
BH_Mapped	Buffer is a valid buffer mapped to an on-disk block.
BH_New	Buffer is newly mapped via get_block() and not yet accessed.
BH_Async_Read	Buffer is undergoing asynchronous read I/O via end_buffer_async_read().
BH_Async_Write	Buffer is undergoing asynchronous write I/O via end_buffer_async_write().
BH_Delay	Buffer does not yet have an associated on-disk block (delayed allocation).
BH_Boundary	Buffer forms the boundary of contiguous blocks—the next block is discontinuous.
BH_Write_EIO	Buffer incurred an I/O error on write.
BH_Ordered	Ordered write.
BH_Eopnotsupp	Buffer incurred a "not supported" error.
BH_Unwritten	Space for the buffer has been allocated on disk but the actual data has not yet been written out.
BH_Quiet	Suppress errors for this buffer.

The bh_state_bits enumeration also contains as the last value in the list a BH_PrivateStart flag. This is not a valid state flag but instead corresponds to the first usable bit of which other code can make use. All bit values equal to and greater than BH_PrivateStart are not used by the block I/O layer proper, so these bits are safe to use

by individual drivers who want to store information in the `b_state` field. Drivers can base the bit values of their internal flags off this flag and rest assured that they are not encroaching on an official bit used by the block I/O layer.

The `b_count` field is the buffer's usage count. The value is incremented and decremented by two inline functions, both of which are defined in `<linux/buffer_head.h>`:

```
static inline void get_bh(struct buffer_head *bh)
{
        atomic_inc(&bh->b_count);
}

static inline void put_bh(struct buffer_head *bh)
{
        atomic_dec(&bh->b_count);
}
```

Before manipulating a buffer head, you must increment its reference count via `get_bh()` to ensure that the buffer head is not deallocated out from under you. When finished with the buffer head, decrement the reference count via `put_bh()`.

The physical block on disk to which a given buffer corresponds is the `b_blocknr`-th logical block on the block device described by `b_bdev`.

The physical page in memory to which a given buffer corresponds is the page pointed to by `b_page`. More specifically, `b_data` is a pointer directly to the block (that exists somewhere in `b_page`), which is `b_size` bytes in length. Therefore, the block is located in memory starting at address `b_data` and ending at address `(b_data + b_size)`.

The purpose of a buffer head is to describe this mapping between the on-disk block and the physical in-memory buffer (which is a sequence of bytes on a specific page). Acting as a descriptor of this buffer-to-block mapping is the data structure's only role in the kernel.

Before the 2.6 kernel, the buffer head was a much more important data structure: It was *the* unit of I/O in the kernel. Not only did the buffer head describe the disk-block-to-physical-page mapping, but it also acted as the container used for all block I/O. This had two primary problems. First, the buffer head was a large and unwieldy data structure (it has shrunken a bit nowadays), and it was neither clean nor simple to manipulate data in terms of buffer heads. Instead, the kernel prefers to work in terms of pages, which are simple and enable for greater performance. A large buffer head describing each individual buffer (which might be smaller than a page) was inefficient. Consequently, in the 2.6 kernel, much work has gone into making the kernel work directly with pages and address spaces instead of buffers. Some of this work is discussed in Chapter 16, "The Page Cache and Page Writeback," where the `address_space` structure and the *pdflush* daemons are discussed.

The second issue with buffer heads is that they describe only a single buffer. When used as the container for all I/O operations, the buffer head forces the kernel to break up potentially large block I/O operations (say, a write) into multiple `buffer_head` structures.

This results in needless overhead and space consumption. As a result, the primary goal of the 2.5 development kernel was to introduce a new, flexible, and lightweight container for block I/O operations. The result is the `bio` structure, which is discussed in the next section.

The `bio` Structure

The basic container for block I/O within the kernel is the `bio` structure, which is defined in `<linux/bio.h>`. This structure represents block I/O operations that are in flight (active) as a list of *segments*. A segment is a chunk of a buffer that is contiguous in memory. Thus, individual buffers need not be contiguous in memory. By allowing the buffers to be described in chunks, 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. Vector I/O such as this is called *scatter-gather I/O*.

Here is `struct bio`, defined in `<linux/bio.h>`, with comments added for each field:

```
struct bio {
        sector_t             bi_sector;         /* associated sector on disk */
        struct bio           *bi_next;          /* list of requests */
        struct block_device  *bi_bdev;          /* associated block device */
        unsigned long        bi_flags;          /* status and command flags */
        unsigned long        bi_rw;             /* read or write? */
        unsigned short       bi_vcnt;           /* number of bio_vecs off */
        unsigned short       bi_idx;            /* current index in bi_io_vec */
        unsigned short       bi_phys_segments;  /* number of segments */
        unsigned int         bi_size;           /* I/O count */
        unsigned int         bi_seg_front_size; /* size of first segment */
        unsigned int         bi_seg_back_size;  /* size of last segment */
        unsigned int         bi_max_vecs;       /* maximum bio_vecs possible */
        unsigned int         bi_comp_cpu;       /* completion CPU */
        atomic_t             bi_cnt;            /* usage counter */
        struct bio_vec       *bi_io_vec;        /* bio_vec list */
        bio_end_io_t         *bi_end_io;        /* I/O completion method */
        void                 *bi_private;       /* owner-private method */
        bio_destructor_t     *bi_destructor;    /* destructor method */
        struct bio_vec       bi_inline_vecs[0]; /* inline bio vectors */
};
```

The primary purpose of a `bio` structure is to represent an in-flight block I/O operation. To this end, the majority of the fields in the structure are housekeeping related. The most important fields are `bi_io_vec`, `bi_vcnt`, and `bi_idx`. Figure 14.2 shows the relationship between the `bio` structure and its friends.

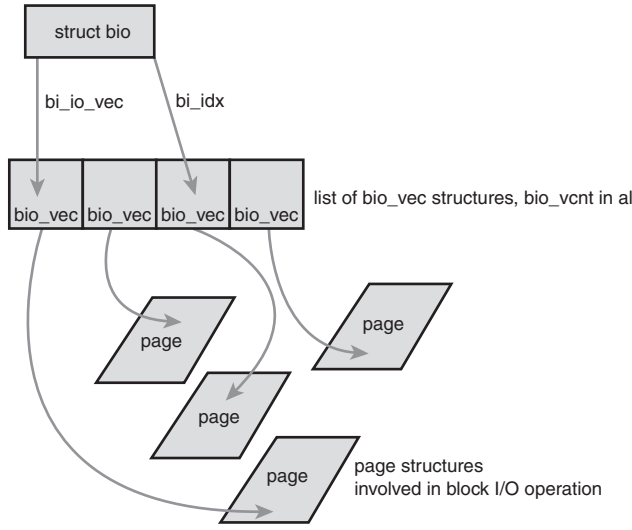

Figure 14.2 Relationship between **`struct bio`**, **`struct bio_vec`**, and **`struct page`**.

I/O vectors

The `bi_io_vec` field points to an array of `bio_vec` structures. These structures are used as lists of individual segments in this specific block I/O operation. Each `bio_vec` is treated as a vector of the form `<page, offset, len>`, which describes a specific segment: the physical page on which it lies, the location of the block as an offset into the page, and the length of the block starting from the given offset. The full array of these vectors describes the entire buffer. The `bio_vec` structure is defined in `<linux/bio.h>`:

```
struct bio_vec {
        /* pointer to the physical page on which this buffer resides */
        struct page     *bv_page;

        /* the length in bytes of this buffer */
        unsigned int    bv_len;

        /* the byte offset within the page where the buffer resides */
        unsigned int    bv_offset;
};
```

In each given block I/O operation, there are `bi_vcnt` vectors in the `bio_vec` array starting with `bi_io_vec`. As the block I/O operation is carried out, the `bi_idx` field is used to point to the current index into the array.

In summary, each block I/O request is represented by a `bio` structure. Each request is composed of one or more blocks, which are stored in an array of `bio_vec` structures.

These structures act as vectors and describe each segment's location in a physical page in memory. The first segment in the I/O operation is pointed to by `b_io_vec`. Each additional segment follows after the first, for a total of `bi_vcnt` segments in the list. As the block I/O layer submits segments in the request, the `bi_idx` field is updated to point to the current segment.

The `bi_idx` field is used to point to the current `bio_vec` in the list, which helps the block I/O layer keep track of partially completed block I/O operations. A more important usage, however, is to allow the splitting of `bio` structures. With this feature, drivers implementing a Redundant Array of Inexpensive Disks (RAID, a hard disk setup that enables single volumes to span multiple disks for performance and reliability purposes) can take a single `bio` structure, initially intended for a single device and split it among the multiple hard drives in the RAID array. All the RAID driver needs to do is copy the `bio` structure and update the `bi_idx` field to point to where the individual drive should start its operation.

The `bio` structure maintains a usage count in the `bi_cnt` field. When this field reaches zero, the structure is destroyed and the backing memory is freed. The following two functions manage the usage counters for you.

```
void bio_get(struct bio *bio)
void bio_put(struct bio *bio)
```

The former increments the usage count, whereas the latter decrements the usage count (and, if the count reaches zero, destroys the `bio` structure). Before manipulating an in-flight `bio` structure, be sure to increment its usage count to make sure it does not complete and deallocate out from under you. When you finish, decrement the usage count in turn.

Finally, the `bi_private` field is a private field for the owner (that is, creator) of the structure. As a rule, you can read or write this field only if you allocated the `bio` structure.

The Old Versus the New

The difference between buffer heads and the new `bio` structure is important. The `bio` structure represents an I/O operation, which may include one or more pages in memory. On the other hand, the `buffer_head` structure represents a single buffer, which describes a single block on the disk. Because buffer heads are tied to a single disk block in a single page, buffer heads result in the unnecessary dividing of requests into block-sized chunks, only to later reassemble them. Because the `bio` structure is lightweight, it can describe discontiguous blocks and does not unnecessarily split I/O operations.

Switching from `struct buffer_head` to `struct bio` provided other benefits, as well:

- The `bio` structure can easily represent high memory, because struct `bio` deals with only physical pages and not direct pointers.
- The `bio` structure can represent both normal page I/O and direct I/O (I/O operations that do not go through the page cache—see Chapter 16, "The Page Cache and Page Writeback," for a discussion on the page cache).

- The `bio` structure makes it easy to perform scatter-gather (vectored) block I/O operations, with the data involved in the operation originating from multiple physical pages.
- The `bio` structure is much more lightweight than a buffer head because it contains only the minimum information needed to represent a block I/O operation and not unnecessary information related to the buffer itself.

The concept of buffer heads is still required, however; buffer heads function as descriptors, mapping disk blocks to pages. The `bio` structure does not contain any information about the state of a buffer—it is simply an array of vectors describing one or more segments of data for a single block I/O operation, plus related information. In the current setup, the `buffer_head` structure is still needed to contain information about buffers while the `bio` structure describes in-flight I/O. Keeping the two structures separate enables each to remain as small as possible.

Request Queues

Block devices maintain *request queues* to store their pending block I/O requests. The request queue is represented by the `request_queue` structure and is defined in `<linux/blkdev.h>`. The request queue contains a doubly linked list of requests and associated control information. Requests are added to the queue by higher-level code in the kernel, such as filesystems. 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. Each item in the queue's request list is a single request, of type `struct request`.

Individual requests on the queue are represented by `struct request`, which is also defined in `<linux/blkdev.h>`. Each request can be composed of more than one `bio` structure because individual requests can operate on multiple consecutive disk blocks. Note that although the blocks on the disk must be adjacent, the blocks in memory need not be; each `bio` structure can describe multiple segments (recall, segments are contiguous chunks of a block in memory) and the request can be composed of multiple `bio` structures.

I/O Schedulers

Simply sending out requests to the block devices in the order that the kernel issues them, as soon as it issues them, results in poor performance. One of the slowest operations in a modern computer is disk seeks. Each seek—positioning the hard disk's head at the location of a specific block—takes many milliseconds. Minimizing seeks is absolutely crucial to the system's performance.

Therefore, the kernel does not issue block I/O requests to the disk in the order they are received or as soon as they are received. Instead, it performs operations called *merging*

and *sorting* to greatly improve the performance of the system as a whole.[2] The subsystem of the kernel that performs these operations is called the *I/O scheduler*.

The I/O scheduler divides the resource of disk I/O among the pending block I/O requests in the system. It does this through the merging and sorting of pending requests in the request queue. The I/O scheduler is not to be confused with the process scheduler (see Chapter 4, "Process Scheduling"), which divides the resource of the processor among the processes on the system. The two subsystems are similar in nature but not the same. Both the process scheduler and the I/O scheduler virtualize a resource among multiple objects. In the case of the process scheduler, the processor is virtualized and shared among the processes on the system. This provides the illusion of virtualization inherent in a multitasking and timesharing operating system, such as any Unix. On the other hand, the I/O scheduler virtualizes block devices among multiple outstanding block requests. This is done to minimize disk seeks and ensure optimum disk performance.

The Job of an I/O Scheduler

An I/O scheduler works by managing a block device's request queue. It decides the order of requests in the queue and at what time each request is dispatched to the block device. It manages the request queue with the goal of reducing seeks, which results in greater *global throughput*. The modifier "global" here is important. An I/O scheduler, very openly, is unfair to some requests at the expense of improving the *overall* performance of the system.

I/O schedulers perform two primary actions to minimize seeks: merging and sorting. Merging is the coalescing of two or more requests into one. Consider an example request that is submitted to the queue by a filesystem—say, to read a chunk of data from a file. (At this point, of course, everything occurs in terms of sectors and blocks and not files but presume that the requested blocks originate from a chunk of a file.) If a request is already in the queue to read from an adjacent sector on the disk (for example, an earlier chunk of the same file), the two requests can be merged into a single request operating on one or more adjacent on-disk sectors. By merging requests, the I/O scheduler reduces the overhead of multiple requests down to a single request. More important only a single command needs to be issued to the disk and servicing the multiple requests can be done without seeking. Consequently, merging requests reduces overhead and minimizes seeks.

Now, assume your fictional read request is submitted to the request queue, but there is no read request to an adjacent sector. You therefore cannot merge this request with any other request. Now, you could simply stick this request onto the tail of the queue. But, what if there are other requests to a similar location on the disk? Would it not make sense to insert this new request into the queue at a spot near other requests operating on physi-

---

[2] *This point must be stressed. A system without these features, or wherein these features are poorly implemented, would perform poorly even with only a modest number of block I/O operations.*

cally near sectors? In fact, I/O schedulers do exactly this. The entire request queue is kept sorted, sectorwise, so that all seeking activity along the queue moves (as much as possible) sequentially over the sectors of the hard disk. The goal is not just to minimize each individual seek but to minimize all seeking by keeping the disk head moving in a straight line. This is similar to the algorithm employed in elevators—elevators do not jump all over, wildly, from floor to floor. Instead, they try to move gracefully in a single direction. When the final floor is reached in one direction, the elevator can reverse course and move in the other direction. Because of this similarity, I/O schedulers (or sometimes just their sorting algorithm) are called *elevators*.

The Linus Elevator

Now let's look at some real-life I/O schedulers. The first I/O scheduler is called the *Linus Elevator*. (Yes, Linus has an elevator named after him!) It was the default I/O scheduler in 2.4. In 2.6, it was replaced by the following I/O schedulers that we will look at—however, because this elevator is simpler than the subsequent ones, while performing many of the same functions, it serves as an excellent introduction.

The Linus Elevator performs both merging and sorting. 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. The Linus Elevator I/O scheduler performs both *front* and *back merging*. The type of merging performed depends on the location of the existing adjacent request. If the new request immediately proceeds an existing request, it is front merged. Conversely, if the new request immediately precedes an existing request, it is back merged. Because of the way files are laid out (usually by increasing sector number) and the I/O operations performed in a typical workload (data is normally read from start to finish and not in reverse), front merging is rare compared to back merging. Nonetheless, the Linus Elevator checks for and performs both types of merge.

If the merge attempt fails, a possible insertion point in the queue (a location in the queue where the new request fits sectorwise between the existing requests) is then sought. If one is found, the new request is inserted there. If a suitable location is not found, the request is added to the tail of the queue. Additionally, if an existing request is found in the queue that is older than a predefined threshold, the new request is added to the tail of the queue even if it can be insertion sorted elsewhere. This prevents many requests to nearby on-disk locations from indefinitely starving requests to other locations on the disk. Unfortunately, this "age" check is not efficient. It does not provide any real attempt to service requests in a given timeframe; it merely stops insertion-sorting requests after a suitable delay. This improves latency but can still lead to request starvation, which was the big must-fix of the 2.4 I/O scheduler.

In summary, when a request is added to the queue, four operations are possible. In order, they are

1. If a request to an adjacent on-disk sector is in the queue, the existing request and the new request merge into a single request.

2. If a request in the queue is sufficiently old, the new request is inserted at the tail of the queue to prevent starvation of the other, older, requests.
3. 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.
4. Finally, if no such suitable insertion point exists, the request is inserted at the tail of the queue.

The Linus elevator is implemented in `block/elevator.c`.

The Deadline I/O Scheduler

The Deadline I/O scheduler sought to prevent the starvation caused by the Linus Elevator. In the interest of minimizing seeks, heavy disk I/O operations to one area of the disk can indefinitely starve request operations to another part of the disk. Indeed, a stream of requests to the same area of the disk can result in other far-off requests never being serviced. This starvation is unfair.

Worse, the general issue of request starvation introduces a specific instance of the problem known as *writes starving reads*. Write operations can usually be committed to disk whenever the kernel gets around to them, entirely asynchronous with respect to the submitting application. Read operations are quite different. Normally, when an application submits a read request, the application blocks until the request is fulfilled. That is, read requests occur synchronously with respect to the submitting application. Although system response is largely unaffected by write latency (the time required to commit a write request), read latency (the time required to commit a read request) is important. Write latency has little bearing on application performance,[3] but an application must wait, twiddling its thumbs, for the completion of each read request. Consequently, read latency is important to the performance of the system.

Compounding the problem, read requests tend to be dependent on each other. For example, consider the reading of a large number of files. Each read occurs in small buffered chunks. The application does not start reading the next chunk (or the next file, for that matter) until the previous chunk is read from disk and returned to the application. Worse, both read and write operations require the reading of various metadata, such as inodes. Reading these blocks off the disk further serializes I/O. Consequently, if each read request is individually starved, the total delay to such applications compounds and can grow enormous. Recognizing that the asynchrony and interdependency of read requests results in a much stronger bearing of read latency on the performance of the system, the Deadline I/O scheduler implements several features to ensure that request starvation in general, and read starvation in specific, is minimized.

---

[3] *We still do not want to delay write requests indefinitely, however, because the kernel wants to ensure that data is eventually written to disk to prevent in-memory buffers from growing too large or too old.*

Note that reducing request starvation comes at a cost to global throughput. Even the Linus Elevator makes this compromise, albeit in a much milder manner. The Linus Elevator could provide better overall throughput (via a greater minimization of seeks) if it *always* inserted requests into the queue sectorwise and never checked for old requests and reverted to insertion at the tail of the queue. Although minimizing seeks is important, indefinite starvation is not good either. The Deadline I/O scheduler, therefore, works harder to limit starvation while still providing good global throughput. Make no mistake: It is a tough act to provide request fairness, yet maximize global throughput.

In the Deadline I/O scheduler, each request is associated with an expiration time. By default, the expiration time is 500 milliseconds in the future for read requests and 5 seconds in the future for write requests. The Deadline I/O scheduler operates similarly to the Linus Elevator in that it maintains a request queue sorted by physical location on disk. It calls this queue the *sorted queue*. When a new request is submitted to the sorted queue, the Deadline I/O scheduler performs merging and insertion like the Linus Elevator.[4] The Deadline I/O scheduler also, however, inserts the request into a second queue that depends on the type of request. Read requests are sorted into a special read FIFO queue, and write requests are inserted into a special write FIFO queue. Although the normal queue is sorted by on-disk sector, these queues are kept FIFO. (Effectively, they are sorted by time.) Consequently, new requests are always added to the tail of the queue. Under normal operation, the Deadline I/O scheduler pulls requests from the head of the sorted queue into the dispatch queue. The dispatch queue is then fed to the disk drive. This results in minimal seeks.

If the request at the head of either the write FIFO queue or the read FIFO queue expires (that is, if the current time becomes greater than the expiration time associated with the request), the Deadline I/O scheduler then begins servicing requests from the FIFO queue. In this manner, the Deadline I/O scheduler attempts to ensure that no request is outstanding longer than its expiration time. See Figure 14.3.

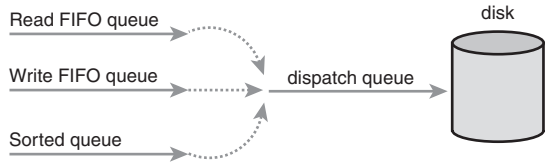

Figure 14.3 The three queues of the Deadline I/O scheduler.

Note that the Deadline I/O scheduler does not make any strict guarantees over request latency. It is capable, however, of generally committing requests on or before their

[4] *Performing front merging is optional in the Deadline I/O scheduler, however. It is not always worth the trouble because many workloads have few requests that can be front merged.*

expiration. This prevents request starvation. Because read requests are given a substantially smaller expiration value than write requests, the Deadline I/O scheduler also works to ensure that write requests do not starve read requests. This preference toward read requests provides minimized read latency.

The Deadline I/O scheduler lives in `block/deadline-iosched.c`.

The Anticipatory I/O Scheduler

Although the Deadline I/O scheduler does a great job minimizing read latency, it does so at the expense of global throughput. Consider a system undergoing heavy write activity. Every time a read request is submitted, the I/O scheduler quickly rushes to handle the read request. This results in the disk seeking over to where the read is, performing the read operation, and then seeking back to continue the ongoing write operation, repeating this little charade for each read request. The preference toward read requests is a good thing, but the resulting pair of seeks (one to the location of the read request and another back to the ongoing write) is detrimental to global disk throughput. The Anticipatory I/O scheduler aims to continue to provide excellent read latency, but also provide excellent global throughput.

First, the Anticipatory I/O scheduler starts with the Deadline I/O scheduler as its base. Therefore, it is not entirely different. The Anticipatory I/O scheduler implements three queues (plus the dispatch queue) and expirations for each request, just like the Deadline I/O scheduler. The major change is the addition of an *anticipation heuristic*.

The Anticipatory I/O scheduler attempts to minimize the seek storm that accompanies read requests issued during other disk I/O activity. When a read request is issued, it is handled as usual, within its usual expiration period. After the request is submitted, however, the Anticipatory I/O scheduler does not immediately seek back and return to handling other requests. Instead, it does absolutely nothing for a few milliseconds. (The actual value is configurable; by default it is six milliseconds.) In those few milliseconds, there is a good chance that the application will submit another read request. Any requests issued to an adjacent area of the disk are immediately handled. After the waiting period elapses, the Anticipatory I/O scheduler seeks back to where it left off and continues handling the previous requests.

It is important to note that the few milliseconds spent in *anticipation* for more requests are well worth it if they minimize even a modest percentage of the back-and-forth seeking that results from the servicing of read requests during other heavy requests. If an adjacent I/O request is issued within the waiting period, the I/O scheduler just saved a pair of seeks. As more and more reads are issued to the same area of disk, many more seeks are prevented.

Of course, if no activity occurs within the waiting period, the Anticipatory I/O scheduler loses, and a few milliseconds are wasted. The key to reaping maximum benefit from the Anticipatory I/O scheduler is correctly anticipating the actions of applications and filesystems. This is done via a set of statistics and associated heuristics. The Anticipatory I/O scheduler keeps track of per-process statistics pertaining to block I/O habits in hopes

of correctly anticipating the actions of applications. With a sufficiently high percentage of correct anticipations, the Anticipatory I/O scheduler can greatly reduce the penalty of seeking to service read requests, while still providing the attention to such requests that system response requires. This enables the Anticipatory I/O scheduler to minimize read latency, while also minimizing the number and duration of seeks. This results in low system latency and high system throughput.

The Anticipatory I/O scheduler lives in the file `block/as-iosched.c` in the kernel source tree. It performs well across most workloads. It is ideal for servers, although it performs poorly on certain uncommon but critical workloads involving seek-happy databases.

The Complete Fair Queuing I/O Scheduler

The Complete Fair Queuing (CFQ) I/O scheduler is an I/O scheduler designed for specialized workloads, but that in practice actually provides good performance across multiple workloads. It is fundamentally different from the previous I/O schedulers that have been covered, however.

The CFQ I/O scheduler assigns incoming I/O requests to specific queues based on the process originating the I/O request. For example, I/O requests from process foo go in foo's queues, and I/O requests from process bar go in bar's queue. Within each queue, requests are coalesced with adjacent requests and insertion sorted. The queues are thus kept sorted sectorwise, as with the other I/O scheduler's queues. The difference with the CFQ I/O scheduler is that there is one queue for each process submitting I/O.

The CFQ I/O scheduler then services the queues round robin, plucking a configurable number of requests (by default, four) from each queue before continuing on to the next. This provides fairness at a per-process level, assuring that each process receives a fair slice of the disk's bandwidth. The intended workload is multimedia, in which such a fair algorithm can guarantee that, for example, an audio player can always refill its audio buffers from disk in time. In practice, however, the CFQ I/O scheduler performs well in many scenarios.

The Complete Fair Queuing I/O scheduler lives in `block/cfq-iosched.c`. It is recommended for desktop workloads, although it performs reasonably well in nearly all workloads without any pathological corner cases. It is now the default I/O scheduler in Linux.

The Noop I/O Scheduler

A fourth and final I/O scheduler is the Noop I/O scheduler, so named because it is basically a noop—it does not do much. The Noop I/O scheduler does not perform sorting or any other form of seek-prevention whatsoever. In turn, it has no need to implement anything akin to the slick algorithms to minimize request latency that you saw in the previous three I/O schedulers.

The Noop I/O scheduler does perform merging, however, as its lone chore. When a new request is submitted to the queue, it is coalesced with any adjacent requests. Other

than this operation, the Noop I/O Scheduler truly is a noop, merely maintaining the request queue in near-FIFO order, from which the block device driver can pluck requests.

The Noop I/O scheduler's lack of hard work is with reason. It is intended for block devices that are truly random-access, such as flash memory cards. If a block device has little or no overhead associated with "seeking," then there is no need for insertion sorting of incoming requests, and the Noop I/O scheduler is the ideal candidate.

The Noop I/O scheduler lives in `block/noop-iosched.c`. It is intended only for random-access devices.

I/O Scheduler Selection

You have now seen four different I/O schedulers in the 2.6 kernel. Each of these I/O schedulers can be enabled and built into the kernel. By default, block devices use the Complete Fair Queuing I/O scheduler. This can be overridden via the boot-time option `elevator=foo` on the kernel command line, where `foo` is a valid and enabled I/O Scheduler. See Table 14.2.

Table 14.2 **Parameters Given to `elevator` Option**

Parameter	I/O Scheduler
`as`	Anticipatory
`cfq`	Complete Fair Queuing
`deadline`	Deadline
`noop`	Noop

For example, the kernel command line option `elevator=as` would enable use of the Anticipatory I/O scheduler for all block devices, overriding the default Complete Fair Queuing scheduler.

Conclusion

In this chapter, we discussed the fundamentals of block devices, and we looked at the data structures used by the block I/O layer: the `bio`, representing in-flight I/O; the `buffer_head`, representing a block-to-page mapping; and the `request` structure, representing a specific I/O request. We followed the I/O request on its brief but important life, culminating in the I/O scheduler. We discussed the dilemmas involved in scheduling I/O and went over the four I/O schedulers currently in the Linux kernel, and the old Linus Elevator from 2.4.

Next up, we tackle the process address space.

15

The Process Address Space

Chapter 12, "Memory Management," looked at how the kernel manages physical memory. In addition to managing its own memory, the kernel also has to manage the memory of user-space processes. This memory is called the *process address space*, which is the representation of memory given to each user-space process on the system. Linux is a virtual memory operating system, and thus the resource of memory is virtualized among the processes on the system. An individual process's view of memory is as if it alone has full access to the system's physical memory. More important, the address space of even a single process can be much larger than physical memory. This chapter discusses how the kernel manages the process address space.

Address Spaces

The process address space consists of the virtual memory addressable by a process and the addresses within the virtual memory that the process is allowed to use. Each process is given a *flat* 32- or 64-bit address space, with the size depending on the architecture. The term *flat* denotes that the address space exists in a single range. (For example, a 32-bit address space extends from the address 0 to 4294967295.) Some operating systems provide a *segmented address space*, with addresses existing not in a single linear range, but instead in multiple segments. Modern virtual memory operating systems generally have a flat memory model and not a segmented one. Normally, this flat address space is unique to each process. A memory address in one process's address space is completely unrelated to that same memory address in another process's address space. Both processes can have different data at the same address in their respective address spaces. Alternatively, processes can elect to share their address space with other processes. We know these processes as *threads*.

A memory address is a given value within the address space, such as `4021f000`. This particular value identifies a specific byte in a process's 32-bit address space. Although a process can address up to 4GB of memory (with a 32-bit address space), it doesn't have permission to access all of it. The interesting part of the address space is the intervals of memory addresses, such as `08048000-0804c000`, that the process has permission to access.

These intervals of legal addresses are called *memory areas*. The process, through the kernel, can dynamically add and remove memory areas to its address space.

The process can access a memory address only in a valid memory area. Memory areas have associated permissions, such as readable, writable, and executable, that the associated process must respect. If a process accesses a memory address not in a valid memory area, or if it accesses a valid area in an invalid manner, the kernel kills the process with the dreaded "Segmentation Fault" message.

Memory areas can contain all sorts of goodies, such as

- A memory map of the executable file's code, called the *text section*.
- A memory map of the executable file's initialized global variables, called the *data section*.
- A memory map of the zero page (a page consisting of all zeros, used for purposes such as this) containing uninitialized global variables, called the *bss section*.[1]
- A memory map of the zero page used for the process's user-space stack. (Do not confuse this with the process's kernel stack, which is separate and maintained and used by the kernel.)
- An additional text, data, and bss section for each shared library, such as the C library and dynamic linker, loaded into the process's address space.
- Any memory mapped files.
- Any shared memory segments.
- Any anonymous memory mappings, such as those associated with `malloc()`.[2]

All valid addresses in the process address space exist in exactly one area; memory areas do not overlap. As you can see, there is a separate memory area for each different chunk of memory in a running process: the stack, object code, global variables, mapped file, and so on.

The Memory Descriptor

The kernel represents a process's address space with a data structure called the *memory descriptor*. This structure contains all the information related to the process address space. The memory descriptor is represented by `struct mm_struct` and defined in

---

[1] *The term "BSS" is historical. It stands for* block started by symbol. *Uninitialized variables are not stored in the executable object because they do not have any associated value. But the C standard decrees that uninitialized global variables are assigned certain default values (basically, all zeros), so the kernel loads the variables (without value) from the executable into memory and maps the zero page over the area, thereby giving the variables the value zero, without having to waste space in the object file with explicit initializations.*

[2] *Newer versions of glibc implement* `malloc()` *via* `mmap()`, *in addition to* `brk()`.

<linux/mm_types.h>. Let's look at the memory descriptor, with comments added describing each field:

```
struct mm_struct {
        struct vm_area_struct  *mmap;               /* list of memory areas */
        struct rb_root         mm_rb;               /* red-black tree of VMAs */
        struct vm_area_struct  *mmap_cache;         /* last used memory area */
        unsigned long          free_area_cache;     /* 1st address space hole */
        pgd_t                  *pgd;                /* page global directory */
        atomic_t               mm_users;            /* address space users */
        atomic_t               mm_count;            /* primary usage counter */
        int                    map_count;           /* number of memory areas */
        struct rw_semaphore    mmap_sem;            /* memory area semaphore */
        spinlock_t             page_table_lock;     /* page table lock */
        struct list_head       mmlist;              /* list of all mm_structs */
        unsigned long          start_code;          /* start address of code */
        unsigned long          end_code;            /* final address of code */
        unsigned long          start_data;          /* start address of data */
        unsigned long          end_data;            /* final address of data */
        unsigned long          start_brk;           /* start address of heap */
        unsigned long          brk;                 /* final address of heap */
        unsigned long          start_stack;         /* start address of stack */
        unsigned long          arg_start;           /* start of arguments */
        unsigned long          arg_end;             /* end of arguments */
        unsigned long          env_start;           /* start of environment */
        unsigned long          env_end;             /* end of environment */
        unsigned long          rss;                 /* pages allocated */
        unsigned long          total_vm;            /* total number of pages */
        unsigned long          locked_vm;           /* number of locked pages */
        unsigned long          saved_auxv[AT_VECTOR_SIZE]; /* saved auxv */
        cpumask_t              cpu_vm_mask;         /* lazy TLB switch mask */
        mm_context_t           context;             /* arch-specific data */
        unsigned long          flags;               /* status flags */
        int                    core_waiters;        /* thread core dump waiters */
        struct core_state      *core_state;         /* core dump support */
        spinlock_t             ioctx_lock;          /* AIO I/O list lock */
        struct hlist_head      ioctx_list;          /* AIO I/O list */
};
```

The mm_users field is the number of processes using this address space. For example, if two threads share this address space, mm_users is equal to two. The mm_count field is the primary reference count for the mm_struct. All mm_users equate to one increment of mm_count. Thus, in the previous example, mm_count is only one. If nine threads shared an address space, mm_users would be nine, but again mm_count would be only one. Only when mm_users reaches zero (when all threads using an address space exit) is mm_count decremented. When mm_count finally reaches zero, there are no remaining references to

this mm_struct, and it is freed. When the kernel operates on an address space and needs to bump its associated reference count, the kernel increments mm_count. Having two counters enables the kernel to differentiate between the main usage counter (mm_count) and the number of processes using the address space (mm_users).

The mmap and mm_rb fields are different data structures that contain the same thing: all the memory areas in this address space. The former stores them in a linked list, whereas the latter stores them in a red-black tree. A red-black tree is a type of binary tree; like all binary trees, searching for a given element is an O(log n) operation. For further discussion on red-black trees, see "Lists and Trees of Memory Areas," later in this chapter.

Although the kernel would normally avoid the extra baggage of using two data structures to organize the same data, the redundancy comes in handy here. The mmap data structure, as a linked list, allows for simple and efficient traversing of all elements. On the other hand, the mm_rb data structure, as a red-black tree, is more suitable to searching for a given element. Memory areas are discussed in more detail later in this chapter. The kernel isn't duplicating the mm_struct structures; just the containing objects. Overlaying a linked list onto a tree, and using both to access the same set of data, is sometimes called a *threaded tree*.

All of the mm_struct structures are strung together in a doubly linked list via the mmlist field. The initial element in the list is the init_mm memory descriptor, which describes the address space of the init process. The list is protected from concurrent access via the mmlist_lock, which is defined in kernel/fork.c.

Allocating a Memory Descriptor

The memory descriptor associated with a given task is stored in the mm field of the task's process descriptor. (The process descriptor is represented by the task_struct structure, defined in <linux/sched.h>.) Thus, current->mm is the current process's memory descriptor. The copy_mm() function copies a parent's memory descriptor to its child during fork(). The mm_struct structure is allocated from the mm_cachep slab cache via the allocate_mm() macro in kernel/fork.c. Normally, each process receives a unique mm_struct and thus a unique process address space.

Processes may elect to share their address spaces with their children by means of the CLONE_VM flag to clone(). The process is then called a *thread*. Recall from Chapter 3, "Process Management," that this is essentially the *only* difference between normal processes and so-called threads in Linux; the Linux kernel does not otherwise differentiate between them. Threads are regular processes to the kernel that merely share certain resources.

In the case that CLONE_VM is specified, allocate_mm() is *not* called, and the process's mm field is set to point to the memory descriptor of its parent via this logic in copy_mm():

```
if (clone_flags & CLONE_VM) {
        /*
         * current is the parent process and
         * tsk is the child process during a fork()
```

```
        */
        atomic_inc(&current->mm->mm_users);
        tsk->mm = current->mm;
}
```

Destroying a Memory Descriptor

When the process associated with a specific address space exits, the `exit_mm()`, defined in `kernel/exit.c`, function is invoked. This function performs some housekeeping and updates some statistics. It then calls `mmput()`, which decrements the memory descriptor's `mm_users` user counter. If the user count reaches zero, `mmdrop()` is called to decrement the `mm_count` usage counter. If *that* counter is finally zero, the `free_mm()` macro is invoked to return the `mm_struct` to the `mm_cachep` slab cache via `kmem_cache_free()`, because the memory descriptor does not have any users.

The `mm_struct` and Kernel Threads

Kernel threads do not have a process address space and therefore do not have an associated memory descriptor. Thus, the `mm` field of a kernel thread's process descriptor is `NULL`. This is the *definition* of a kernel thread—processes that have no user context.

This lack of an address space is fine because kernel threads do not ever access any user-space memory. (Whose would they access?) Because kernel threads do not have any pages in user-space, they do not deserve their own memory descriptor and page tables. (Page tables are discussed later in the chapter.) Despite this, kernel threads need some of the data, such as the page tables, even to access kernel memory. To provide kernel threads the needed data, without wasting memory on a memory descriptor and page tables, or wasting processor cycles to switch to a new address space whenever a kernel thread begins running, kernel threads use the memory descriptor of whatever task ran previously.

Whenever a process is scheduled, the process address space referenced by the process's `mm` field is loaded. The `active_mm` field in the process descriptor is then updated to refer to the new address space. Kernel threads do not have an address space and `mm` is `NULL`. Therefore, when a kernel thread is scheduled, the kernel notices that `mm` is `NULL` and keeps the previous process's address space loaded. The kernel then updates the `active_mm` field of the kernel thread's process descriptor to refer to the previous process's memory descriptor. The kernel thread can then use the previous process's page tables as needed. Because kernel threads do not access user-space memory, they make use of only the information in the address space pertaining to kernel memory, which is the same for all processes.

Virtual Memory Areas

The memory area structure, `vm_area_struct`, represents memory areas. It is defined in `<linux/mm_types.h>`. In the Linux kernel, memory areas are often called *virtual memory areas* (abbreviated *VMAs*).

The vm_area_struct structure describes a single memory area over a contiguous interval in a given address space. The kernel treats each memory area as a unique memory object. Each memory area possesses certain properties, such as permissions and a set of associated operations. In this manner, each VMA structure can represent different types of memory areas—for example, memory-mapped files or the process's user-space stack. This is similar to the object-oriented approach taken by the VFS layer (see Chapter 13). Here's the structure, with comments added describing each field:

```
struct vm_area_struct {
        struct mm_struct             *vm_mm;        /* associated mm_struct */
        unsigned long                vm_start;      /* VMA start, inclusive */
        unsigned long                vm_end;        /* VMA end , exclusive */
        struct vm_area_struct        *vm_next;      /* list of VMA's */
        pgprot_t                     vm_page_prot;  /* access permissions */
        unsigned long                vm_flags;      /* flags */
        struct rb_node               vm_rb;         /* VMA's node in the tree */
        union {         /* links to address_space->i_mmap or i_mmap_nonlinear */
                struct {
                        struct list_head        list;
                        void                    *parent;
                        struct vm_area_struct   *head;
                } vm_set;
                struct prio_tree_node prio_tree_node;
        } shared;
        struct list_head             anon_vma_node;     /* anon_vma entry */
        struct anon_vma              *anon_vma;         /* anonymous VMA object */
        struct vm_operations_struct  *vm_ops;           /* associated ops */
        unsigned long                vm_pgoff;          /* offset within file */
        struct file                  *vm_file;          /* mapped file, if any */
        void                         *vm_private_data;  /* private data */
};
```

Recall that each memory descriptor is associated with a unique interval in the process's address space. The vm_start field is the initial (lowest) address in the interval, and the vm_end field is the first byte after the final (highest) address in the interval. That is, vm_start is the inclusive start, and vm_end is the exclusive end of the memory interval. Thus, vm_end - vm_start is the length in bytes of the memory area, which exists over the interval [vm_start, vm_end). Intervals in different memory areas in the same address space cannot overlap.

The vm_mm field points to this VMA's associated mm_struct. Note that each VMA is unique to the mm_struct with which it is associated. Therefore, even if two separate processes map the same file into their respective address spaces, each has a unique vm_area_struct to identify its unique memory area. Conversely, two threads that share an address space also share all the vm_area_struct structures therein.

VMA Flags

The vm_flags field contains bit flags, defined in <linux/mm.h>, that specify the behavior of and provide information about the pages contained in the memory area. Unlike permissions associated with a specific physical page, the VMA flags specify behavior for which the kernel is responsible, not the hardware. Furthermore, vm_flags contains information that relates to each page in the memory area, or the memory area as a whole, and not specific individual pages. Table 15.1 is a listing of the possible vm_flags values.

Table 15.1 **vm_flags**

Flag	Effect on the VMA and Its Pages
VM_READ	Pages can be read from.
VM_WRITE	Pages can be written to.
VM_EXEC	Pages can be executed.
VM_SHARED	Pages are shared.
VM_MAYREAD	The VM_READ flag can be set.
VM_MAYWRITE	The VM_WRITE flag can be set.
VM_MAYEXEC	The VM_EXEC flag can be set.
VM_MAYSHARE	The VM_SHARE flag can be set.
VM_GROWSDOWN	The area can grow downward.
VM_GROWSUP	The area can grow upward.
VM_SHM	The area is used for shared memory.
VM_DENYWRITE	The area maps an unwritable file.
VM_EXECUTABLE	The area maps an executable file.
VM_LOCKED	The pages in this area are locked.
VM_IO	The area maps a device's I/O space.
VM_SEQ_READ	The pages seem to be accessed sequentially.
VM_RAND_READ	The pages seem to be accessed randomly.
VM_DONTCOPY	This area must not be copied on fork().
VM_DONTEXPAND	This area cannot grow via mremap().
VM_RESERVED	This area must not be swapped out.
VM_ACCOUNT	This area is an accounted VM object.
VM_HUGETLB	This area uses hugetlb pages.
VM_NONLINEAR	This area is a nonlinear mapping.

Let's look at some of the more important and interesting flags in depth. The VM_READ, VM_WRITE, and VM_EXEC flags specify the usual read, write, and execute permissions for the pages *in this particular memory area*. They are combined as needed to form the appropriate access permissions that a process accessing this VMA must respect. For example, the object code for a process might be mapped with VM_READ and VM_EXEC but not VM_WRITE. On the other hand, the data section from an executable object would be mapped VM_READ and VM_WRITE, but VM_EXEC would make little sense. Meanwhile, a read-only memory mapped data file would be mapped with only the VM_READ flag.

The VM_SHARED flag specifies whether the memory area contains a mapping that is shared among multiple processes. If the flag is set, it is intuitively called a *shared mapping*. If the flag is not set, only a single process can view this particular mapping, and it is called a *private mapping*.

The VM_IO flag specifies that this memory area is a mapping of a device's I/O space. This field is typically set by device drivers when mmap() is called on their I/O space. It specifies, among other things, that the memory area must not be included in any process's core dump. The VM_RESERVED flag specifies that the memory region must not be swapped out. It is also used by device driver mappings.

The VM_SEQ_READ flag provides a hint to the kernel that the application is performing sequential (that is, linear and contiguous) reads in this mapping. The kernel can then opt to increase the read-ahead performed on the backing file. The VM_RAND_READ flag specifies the exact opposite: that the application is performing relatively random (that is, not sequential) reads in this mapping. The kernel can then opt to decrease or altogether disable read-ahead on the backing file. These flags are set via the madvise() system call with the MADV_SEQUENTIAL and MADV_RANDOM flags, respectively. Read-ahead is the act of reading sequentially ahead of requested data, in hopes that the additional data will be needed soon. Such behavior is beneficial if applications are reading data sequentially. If data access patterns are random, however, read-ahead is not effective.

VMA Operations

The vm_ops field in the vm_area_struct structure points to the table of operations associated with a given memory area, which the kernel can invoke to manipulate the VMA. The vm_area_struct acts as a generic object for representing any type of memory area, and the operations table describes the specific methods that can operate on this particular instance of the object.

The operations table is represented by struct vm_operations_struct and is defined in <linux/mm.h>:

```
struct vm_operations_struct {
        void (*open) (struct vm_area_struct *);
        void (*close) (struct vm_area_struct *);
        int (*fault) (struct vm_area_struct *, struct vm_fault *);
        int (*page_mkwrite) (struct vm_area_struct *vma, struct vm_fault *vmf);
        int (*access) (struct vm_area_struct *, unsigned long ,
                       void *, int, int);
};
```

Here's a description for each individual method:

- `void open(struct vm_area_struct *area)`

  This function is invoked when the given memory area is added to an address space.
- `void close(struct vm_area_struct *area)`

  This function is invoked when the given memory area is removed from an address space.
- `int fault(struct vm_area_sruct *area, struct vm_fault *vmf)`

  This function is invoked by the page fault handler when a page that is not present in physical memory is accessed.
- `int page_mkwrite(struct vm_area_sruct *area, struct vm_fault *vmf)`

  This function is invoked by the page fault handler when a page that was read-only is being made writable.
- `int access(struct vm_area_struct *vma, unsigned long address, void *buf, int len, int write)`

  This function is invoked by `access_process_vm()` when `get_user_pages()` fails.

Lists and Trees of Memory Areas

As discussed, memory areas are accessed via both the `mmap` and the `mm_rb` fields of the memory descriptor. These two data structures independently point to all the memory area objects associated with the memory descriptor. In fact, they both contain pointers to the same `vm_area_struct` structures, merely represented in different ways.

The first field, `mmap`, links together all the memory area objects in a singly linked list. Each `vm_area_struct` structure is linked into the list via its `vm_next` field. The areas are sorted by ascending address. The first memory area is the `vm_area_struct` structure to which `mmap` points. The last structure points to `NULL`.

The second field, `mm_rb`, links together all the memory area objects in a red-black tree. The root of the red-black tree is `mm_rb`, and each `vm_area_struct` structure in this address space is linked to the tree via its `vm_rb` field.

A *red-black tree* is a type of balanced binary tree. Each element in a red-black tree is called a *node*. The initial node is called the *root* of the tree. Most nodes have two children: a left child and a right child. Some nodes have only one child, and the final nodes, called *leaves*, have no children. For any node, the elements to the left are smaller in value, whereas the elements to the right are larger in value. Furthermore, each node is assigned a color (red or black, hence the name of this tree) according to two rules: The children of a red node are black, and every path through the tree from a node to a leaf must contain the same number of black nodes. The root node is always red. Searching of, insertion to, and deletion from the tree is an `O(log(n))` operation.

The linked list is used when every node needs to be traversed. The red-black tree is used when locating a specific memory area in the address space. In this manner, the ker-

nel uses the redundant data structures to provide optimal performance regardless of the operation performed on the memory areas.

Memory Areas in Real Life

Let's look at a particular process's address space and the memory areas inside. This task uses the useful `/proc` filesystem and the `pmap`(1) utility. The example is a simple user-space program, which does absolutely nothing of value, except act as an example:

```
int main(int argc, char *argv[])
{
        return 0;
}
```

Take note of a few of the memory areas in this process's address space. First, you know there is the text section, data section, and bss. Assuming this process is dynamically linked with the C library, these three memory areas also exist for `libc.so` and again for `ld.so`. Finally, there is also the process's stack.

The output from `/proc/<pid>/maps` lists the memory areas in this process's address space:

```
rlove@wolf:~$ cat /proc/1426/maps
00e80000-00faf000 r-xp 00000000 03:01 208530     /lib/tls/libc-2.5.1.so
00faf000-00fb2000 rw-p 0012f000 03:01 208530     /lib/tls/libc-2.5.1.so
00fb2000-00fb4000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 03:03 439029     /home/rlove/src/example
08049000-0804a000 rw-p 00000000 03:03 439029     /home/rlove/src/example
40000000-40015000 r-xp 00000000 03:01 80276      /lib/ld-2.5.1.so
40015000-40016000 rw-p 00015000 03:01 80276      /lib/ld-2.5.1.so
4001e000-4001f000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

The data is in the form

```
start-end permission   offset   major:minor   inode   file
```

The `pmap`(1) utility[3] formats this information in a bit more readable manner:

```
rlove@wolf:~$ pmap 1426
example[1426]
00e80000 (1212 KB)     r-xp (03:01 208530)   /lib/tls/libc-2.5.1.so
00faf000 (12 KB)       rw-p (03:01 208530)   /lib/tls/libc-2.5.1.so
00fb2000 (8 KB)        rw-p (00:00 0)
08048000 (4 KB)        r-xp (03:03 439029)   /home/rlove/src/example
```

[3] *The `pmap`(1) utility displays a formatted listing of a process's memory areas. It is a bit more readable than the `/proc` output, but it is the same information. It is found in newer versions of the procps package.*

```
08049000 (4 KB)        rw-p (03:03 439029)   /home/rlove/src/example
40000000 (84 KB)       r-xp (03:01 80276)    /lib/ld-2.5.1.so
40015000 (4 KB)        rw-p (03:01 80276)    /lib/ld-2.5.1.so
4001e000 (4 KB)        rw-p (00:00 0)
bfffe000 (8 KB)        rwxp (00:00 0)        [ stack ]
mapped: 1340 KB        writable/private: 40 KB     shared: 0 KB
```

The first three rows are the text section, data section, and bss of `libc.so`, the C library. The next two rows are the text and data section of our executable object. The following three rows are the text section, data section, and bss for `ld.so`, the dynamic linker. The last row is the process's stack.

Note how the text sections are all readable and executable, which is what you expect for object code. On the other hand, the data section and bss (which both contain global variables) are marked readable and writable, but not executable. The stack is, naturally, readable, writable, and executable—not of much use otherwise.

The entire address space takes up about 1340KB, but only 40KB are writable and private. If a memory region is shared or nonwritable, the kernel keeps only one copy of the backing file in memory. This might seem like common sense for shared mappings, but the nonwritable case can come as a bit of a surprise. If you consider that a nonwritable mapping can never be changed (the mapping is only read from), it is clear that it is safe to load the image only once into memory. Therefore, the C library needs to occupy only 1212KB in physical memory and not 1212KB multiplied by every process using the library. Because this process has access to about 1340KB worth of data and code, yet consumes only about 40KB of physical memory, the space savings from such sharing is substantial.

Note the memory areas without a mapped file on device `00:00` and inode zero. This is the zero page, which is a mapping that consists of all zeros. By mapping the zero page over a writable memory area, the area is in effect "initialized" to all zeros. This is important in that it provides a zeroed memory area, which is expected by the bss. Because the mapping is not shared, as soon as the process writes to this data, a copy is made (à la copy-on-write) and the value updated from zero.

Each of the memory areas associated with the process corresponds to a `vm_area_struct` structure. Because the process was not a thread, it has a unique `mm_struct` structure referenced from its `task_struct`.

Manipulating Memory Areas

The kernel often has to perform operations on a memory area, such as whether a given address exists in a given VMA. These operations are frequent and form the basis of the `mmap()` routine, which is covered in the next section. A handful of helper functions are defined to assist these jobs.

These functions are all declared in `<linux/mm.h>`.

find_vma()

The kernel provides a function, `find_vma()`, for searching for the VMA in which a given memory address resides. It is defined in `mm/mmap.c`:

```
struct vm_area_struct * find_vma(struct mm_struct *mm, unsigned long addr);
```

This function searches the given address space for the first memory area whose `vm_end` field is greater than `addr`. In other words, this function finds the first memory area that contains `addr` or begins at an address greater than `addr`. If no such memory area exists, the function returns `NULL`. Otherwise, a pointer to the `vm_area_struct` structure is returned. Note that because the returned VMA may start at an address greater than `addr`, the given address does not necessarily lie *inside* the returned VMA. The result of the `find_vma()` function is cached in the `mmap_cache` field of the memory descriptor. Because of the probability of an operation on one VMA being followed by more operations on that same VMA, the cached results have a decent hit rate (about 30–40% in practice). Checking the cached result is quick. If the given address is *not* in the cache, you must search the memory areas associated with this memory descriptor for a match. This is done via the red-black tree:

```
struct vm_area_struct * find_vma(struct mm_struct *mm, unsigned long addr)
{
        struct vm_area_struct *vma = NULL;

        if (mm) {
                vma = mm->mmap_cache;
                if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
                        struct rb_node *rb_node;

                        rb_node = mm->mm_rb.rb_node;
                        vma = NULL;
                        while (rb_node) {
                                struct vm_area_struct * vma_tmp;

                                vma_tmp = rb_entry(rb_node,
                                                   struct vm_area_struct, vm_rb);
                                if (vma_tmp->vm_end > addr) {
                                        vma = vma_tmp;
                                        if (vma_tmp->vm_start <= addr)
                                                break;
                                        rb_node = rb_node->rb_left;
                                } else
                                        rb_node = rb_node->rb_right;
                        }
                        if (vma)
                                mm->mmap_cache = vma;
                }
```

```
        }

        return vma;
}
```

The initial check of `mmap_cache` tests whether the cached VMA contains the desired address. Note that simply checking whether the VMA's `vm_end` field is bigger than `addr` would not ensure that this is the first such VMA that is larger than `addr`. Thus, for the cache to be useful here, the given `addr` must lie in the VMA—thankfully, this is just the sort of scenario in which consecutive operations on the same VMA would occur.

If the cache does not contain the desired VMA, the function must search the red-black tree. If the current VMA's `vm_end` is larger than `addr`, the function follows the left child; otherwise, it follows the right child. The function terminates as soon as a VMA is found that contains `addr`. If such a VMA is not found, the function continues traversing the tree and returns the first VMA it found that starts after `addr`. If no VMA is ever found, `NULL` is returned.

find_vma_prev()

The `find_vma_prev()` function works the same as `find_vma()`, but it also returns the last VMA *before* `addr`. The function is also defined in `mm/mmap.c` and declared in `<linux/mm.h>`:

```
struct vm_area_struct * find_vma_prev(struct mm_struct *mm, unsigned long addr,
                                      struct vm_area_struct **pprev)
```

The `pprev` argument stores a pointer to the VMA preceding `addr`.

find_vma_intersection()

The `find_vma_intersection()` function returns the first VMA that overlaps a given address interval. The function is defined in `<linux/mm.h>` because it is inline:

```
static inline struct vm_area_struct *
find_vma_intersection(struct mm_struct *mm,
                      unsigned long start_addr,
                      unsigned long end_addr)
{
        struct vm_area_struct *vma;

        vma = find_vma(mm, start_addr);
        if (vma && end_addr <= vma->vm_start)
                vma = NULL;
        return vma;
}
```

The first parameter is the address space to search, `start_addr` is the start of the interval, and `end_addr` is the end of the interval.

Obviously, if `find_vma()` returns `NULL`, so would `find_vma_intersection()`. If `find_vma()` returns a valid VMA, however, `find_vma_intersection()` returns the same VMA only if it does *not* start after the end of the given address range. If the returned memory area does start after the end of the given address range, the function returns `NULL`.

`mmap()` and `do_mmap()`: Creating an Address Interval

The `do_mmap()` function is used by the kernel to create a new linear address interval. Saying that this function creates a new VMA is not technically correct, because if the created address interval is adjacent to an existing address interval, and if they share the same permissions, the two intervals are merged into one. If this is not possible, a new VMA is created. In any case, `do_mmap()` is the function used to add an address interval to a process's address space—whether that means expanding an existing memory area or creating a new one.

The `do_mmap()` function is declared in `<linux/mm.h>`:

```
unsigned long do_mmap(struct file *file, unsigned long addr,
                      unsigned long len, unsigned long prot,
                      unsigned long flag, unsigned long offset)
```

This function maps the file specified by `file` at offset `offset` for length `len`. The `file` parameter can be `NULL` and `offset` can be zero, in which case the mapping will not be backed by a file. In that case, this is called an *anonymous mapping*. If a file and offset are provided, the mapping is called a *file-backed mapping*.

The `addr` function optionally specifies the initial address from which to start the search for a free interval.

The `prot` parameter specifies the access permissions for pages in the memory area. The possible permission flags are defined in `<asm/mman.h>` and are unique to each supported architecture, although in practice each architecture defines the flags listed in Table 15.2.

Table 15.2 **Page Protection Flags**

Flag	Effect on the Pages in the New Interval
`PROT_READ`	Corresponds to `VM_READ`
`PROT_WRITE`	Corresponds to `VM_WRITE`
`PROT_EXEC`	Corresponds to `VM_EXEC`
`PROT_NONE`	Cannot access page

The flags parameter specifies flags that correspond to the remaining VMA flags. These flags specify the type and change the behavior of the mapping. They are also defined in <asm/mman.h>. See Table 15.3.

Table 15.3 **Map Type Flags**

Flag	Effect on the New Interval
MAP_SHARED	The mapping can be shared.
MAP_PRIVATE	The mapping cannot be shared.
MAP_FIXED	The new interval *must* start at the given address addr.
MAP_ANONYMOUS	The mapping is not file-backed, but is anonymous.
MAP_GROWSDOWN	Corresponds to VM_GROWSDOWN.
MAP_DENYWRITE	Corresponds to VM_DENYWRITE.
MAP_EXECUTABLE	Corresponds to VM_EXECUTABLE.
MAP_LOCKED	Corresponds to VM_LOCKED.
MAP_NORESERVE	No need to reserve space for the mapping.
MAP_POPULATE	Populate (prefault) page tables.
MAP_NONBLOCK	Do not block on I/O.

If any of the parameters are invalid, do_mmap() returns a negative value. Otherwise, a suitable interval in virtual memory is located. If possible, the interval is merged with an adjacent memory area. Otherwise, a new vm_area_struct structure is allocated from the vm_area_cachep slab cache, and the new memory area is added to the address space's linked list and red-black tree of memory areas via the vma_link() function. Next, the total_vm field in the memory descriptor is updated. Finally, the function returns the initial address of the newly created address interval.

The do_mmap() functionality is exported to user-space via the mmap() system call. The mmap() system call is defined as

```
void  * mmap2(void *start,
              size_t length,
              int prot,
              int flags,
              int fd,
              off_t pgoff)
```

This system call is named mmap2() because it is the second variant of mmap(). The original mmap() took an offset in bytes as the last parameter; the current mmap2() receives the offset in pages. This enables larger files with larger offsets to be mapped. The original

`mmap()`, as specified by POSIX, is available from the C library as `mmap()`, but is no longer implemented in the kernel proper, whereas the new version is available as `mmap2()`. Both library calls use the `mmap2()` system call, with the original `mmap()` converting the offset from bytes to pages.

`munmap()` and `do_munmap()`: Removing an Address Interval

The `do_munmap()` function removes an address interval from a specified process address space. The function is declared in `<linux/mm.h>`:

```
int do_munmap(struct mm_struct *mm, unsigned long start, size_t len)
```

The first parameter specifies the address space from which the interval starting at address `start` of length `len` bytes is removed. On success, zero is returned. Otherwise, a negative error code is returned.

The `munmap()` system call is exported to user-space as a means to enable processes to remove address intervals from their address space; it is the complement of the `mmap()` system call:

```
int munmap(void *start, size_t length)
```

The system call is defined in `mm/mmap.c` and acts as a simple wrapper to `do_munmap()`:

```
asmlinkage long sys_munmap(unsigned long addr, size_t len)
{
        int ret;
        struct mm_struct *mm;

        mm = current->mm;
        down_write(&mm->mmap_sem);
        ret = do_munmap(mm, addr, len);
        up_write(&mm->mmap_sem);
        return ret;
}
```

Page Tables

Although applications operate on virtual memory mapped to physical addresses, processors operate directly on those physical addresses. Consequently, when an application accesses a virtual memory address, it must first be converted to a physical address before the processor can resolve the request. Performing this lookup is done via page tables. Page tables work by splitting the virtual address into chunks. Each chunk is used as an index into a table. The table points to either another table or the associated physical page.

In Linux, the page tables consist of three levels. The multiple levels enable a sparsely populated address space, even on 64-bit machines. If the page tables were implemented as

a single static array, their size on even 32-bit architectures would be enormous. Linux uses three levels of page tables even on architectures that do not support three levels in hardware. (For example, some hardware uses only two levels or implements a hash in hardware.) Using three levels is a sort of "greatest common denominator"—architectures with a less complicated implementation can simplify the kernel page tables as needed with compiler optimizations.

The top-level page table is the page global directory (PGD), which consists of an array of `pgd_t` types. On most architectures, the `pgd_t` type is an `unsigned long`. The entries in the PGD point to entries in the second-level directory, the PMD.

The second-level page table is the page middle directory (PMD), which is an array of `pmd_t` types. The entries in the PMD point to entries in the PTE.

The final level is called simply the page table and consists of page table entries of type `pte_t`. Page table entries point to physical pages.

In most architectures, page table lookups are handled (at least to some degree) by hardware. In normal operation, hardware can handle much of the responsibility of using the page tables. The kernel must set things up, however, in such a way that the hardware is happy and can do its thing. Figure 15.1 diagrams the flow of a virtual to physical address lookup using page tables.

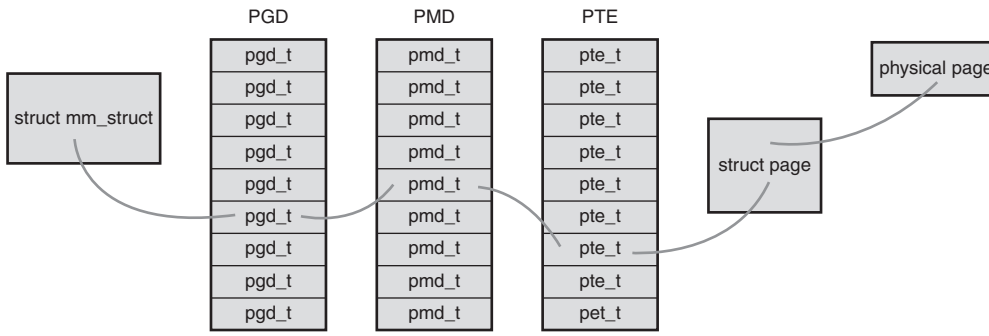

Figure 15.1 Virtual-to-physical address lookup.

Each process has its own page tables (threads share them, of course). The `pgd` field of the memory descriptor points to the process's page global directory. Manipulating and traversing page tables requires the `page_table_lock`, which is located inside the associated memory descriptor.

Page table data structures are quite architecture-dependent and thus are defined in `<asm/page.h>`.

Because nearly every access of a page in virtual memory must be resolved to its corresponding address in physical memory, the performance of the page tables is very critical. Unfortunately, looking up all these addresses in memory can be done only so quickly. To facilitate this, most processors implement a *translation lookaside buffer*, or simply *TLB*,

which acts as a hardware cache of virtual-to-physical mappings. When accessing a virtual address, the processor first checks whether the mapping is cached in the TLB. If there is a hit, the physical address is immediately returned. Otherwise, if there is a miss, the page tables are consulted for the corresponding physical address.

Nonetheless, page table management is still a critical—and evolving—part of the kernel. Changes to this area in 2.6 include allocating parts of the page table out of high memory. Future possibilities include shared page tables with copy-on-write semantics. In that scheme, page tables would be shared between parent and child across a `fork()`. When the parent or the child attempted to modify a particular page table entry, a copy would be created, and the two processes would no longer share that entry. Sharing page tables would remove the overhead of copying the page table entries on `fork()`.

Conclusion

In this suspense-laden chapter, we looked at the abstraction of virtual memory provided to each process. We looked at how the kernel represents the process address space (via `struct mm_struct`) and how the kernel represents regions of memory within that space (`struct vm_area_struct`). We covered how the kernel creates (via `mmap()`) and destroys (via `munmap()`) these memory regions. Finally, we covered page tables. Because Linux is a virtual memory–based operating system, these concepts are essential to its operation and process model.

The next chapter covers the page cache, a general in-memory data cache used to fulfill all page I/O, and how the kernel performs page-based data writeback.

16

The Page Cache and Page Writeback

The Linux kernel implements a disk cache called the *page cache*. The goal of this cache is to minimize disk I/O by storing data in physical memory that would otherwise require disk access. This chapter deals with the page cache and the process by which changes to the page cache are propagated back to disk, which is called *page writeback*.

Two factors comingle to make disk caches a critical component of any modern operating system. First, disk access is several orders of magnitude slower than memory access—milliseconds versus nanoseconds. Accessing data from memory rather than the disk is much faster, and accessing data from the processor's L1 or L2 cache is faster still. Second, data accessed once will, with a high likelihood, find itself accessed again in the near future. This principle—that access to a particular piece of data tends to be clustered in time—is called *temporal locality*, which ensures that if data is cached on its first access, there is a high probability of a cache hit (access to data in the cache) in the near future. Given that memory is so much faster than disk, coupled with the fact that once-used is likely twice-used data, an in-memory cache of the disk is a large performance win.

Approaches to Caching

The page cache consists of physical pages in RAM, the contents of which correspond to physical blocks on a disk. The size of the page cache is dynamic; it can grow to consume any free memory and shrink to relieve memory pressure. We call the storage device being cached the *backing store* because the disk stands behind the cache as the source of the canonical version of any cached data. Whenever the kernel begins a read operation—for example, when a process issues the `read()` system call—it first checks if the requisite data is in the page cache. If it is, the kernel can forgo accessing the disk and read the data directly out of RAM. This is called a *cache hit*. If the data is not in the cache, called a *cache miss*, the kernel must schedule block I/O operations to read the data off the disk. After the data is read off the disk, the kernel populates the page cache with the data so that any subsequent reads can occur out of the cache. Entire files need not be cached; the page

cache can hold some files in their entirety while storing only a page or two of other files. What is cached depends on what has been accessed.

Write Caching

This explains how data ends up in the page cache via read operations, but what happens when a process writes to disk, for example via the `write()` system call? Generally speaking, caches can implement one of three different strategies. In the first strategy, called *no-write*, the cache simply does not cache write operations. A write operation against a piece of data stored in the cache would be written directly to disk, invalidating the cached data and requiring it to be read from disk again on any subsequent read. Caches rarely employ this strategy because it not only fails to cache write operations, but it also makes them costly by invalidating the cache.

In the second strategy, a write operation would automatically update both the in-memory cache and the on-disk file. This approach is called a *write-through cache* because write operations immediately go *through* the cache to the disk. This approach has the benefit of keeping the cache *coherent*—synchronized and valid for the backing store—without needing to *invalidate* it. It is also simple.

The third strategy, employed by Linux, is called *write-back*.[1] In a write-back cache, processes perform write operations directly into the page cache. The backing store is not immediately or directly updated. Instead, the written-to pages in the page cache are marked as *dirty* and are added to a *dirty list*. Periodically, pages in the dirty list are written back to disk in a process called *writeback*, bringing the on-disk copy in line with the in-memory cache. The pages are then marked as no longer dirty. The term "dirty" can be confusing because what is actually dirty is not the data in the page cache (which is up to date) but the data on disk (which is out of date). A better term would be *unsynchronized*. Nonetheless, we say the cache contents, not the invalid disk contents, are dirty. A write-back is generally considered superior to a write-through strategy because by deferring the writes to disk, they can be coalesced and performed in bulk at a later time. The downside is complexity.

Cache Eviction

The final piece to caching is the process by which data is removed from the cache, either to make room for more relevant cache entries or to shrink the cache to make available more RAM for other uses. This process, and the strategy that decides what to remove, is called *cache eviction*. Linux's cache eviction works by selecting *clean* (not dirty) pages and

---

[1] *Some books or operating systems call such a strategy a* copy-back *or* write-behind *cache. All three names are synonyms. Linux and other Unix systems use the noun "write-back" to refer to the caching strategy and the verb "writeback" to refer to the action of writing cached data back to the backing store. This book follows that usage.*

simply replacing them with something else. If insufficient clean pages are in the cache, the kernel forces a writeback to make more clean pages available. The hard part is deciding *what* to evict. The ideal eviction strategy evicts the pages least likely to be used in the future. Of course, knowing what pages are least likely to be accessed requires knowing the future, which is why this hopeful strategy is often referred to as the *clairvoyant algorithm*. Such a strategy is ideal, but impossible to implement.

Least Recently Used

Cache eviction strategies attempt to approximate the clairvoyant algorithm with what information they have access to. One of the more successful algorithms, particularly for general-purpose page caches, is called *least recently used*, or *LRU*. An LRU eviction strategy requires keeping track of when each page is accessed (or at least sorting a list of pages by access time) and evicting the pages with the oldest timestamp (or at the start of the sorted list). This strategy works well because the longer a piece of cached data sits idle, the less likely it is to be accessed in the near future. Least recently used is a great approximation of most likely to be used. However, one particular failure of the LRU strategy is that many files are accessed once and then never again. Putting them at the top of the LRU list is thus not optimal. Of course, as before, the kernel has no way of knowing that a file is going to be accessed only once. But it does know how many times it has been accessed in the past.

The Two-List Strategy

Linux, therefore, implements a modified version of LRU, called the *two-list strategy*. Instead of maintaining one list, the LRU list, Linux keeps two lists: the *active list* and the *inactive list*. Pages on the active list are considered "hot" and are not available for eviction. Pages on the inactive list are available for cache eviction. Pages are placed on the active list only when they are accessed *while already residing* on the inactive list. Both lists are maintained in a pseudo-LRU manner: Items are added to the tail and removed from the head, as with a queue. The lists are kept in balance: If the active list becomes much larger than the inactive list, items from the active list's head are moved back to the inactive list, making them available for eviction. The two-list strategy solves the only-used-once failure in a classic LRU and also enables simpler, pseudo-LRU semantics to perform well. This two-list approach is also known as *LRU/2*; it can be generalized to n-lists, called *LRU/n*.

We now know how the page cache is populated (via reads and writes), how it is synchronized in the face of writes (via writeback), and how old data is evicted to make way for new data (via a two-list strategy). Let's now consider a real-world scenario to see how the page cache benefits the system. Assume you are working on a large software project—the Linux kernel, perhaps—and have many source files open. As you open and read source code, the files are stored in the page cache. Jumping around from file to file is instantaneous as the data is cached. As you edit the files, saving them appears instantaneous as well because the writes only need to go to memory, not the disk. When you compile the project, the cached files enable the compilation to proceed with far fewer disk accesses, and thus much more quickly. If the entire source tree is too big to fit in

memory, some of it must be evicted—and thanks to the two-list strategy, any evicted files will be on the inactive list and likely not one of the source files you are directly editing. Later, hopefully when you are not compiling, the kernel will perform page writeback and update the on-disk copies of the source files with any changes you made. This caching results in a dramatic increase in system performance. To see the difference, compare how long it takes to compile your large software project when "cache cold"—say, fresh off a reboot—versus "cache warm."

The Linux Page Cache

The page cache, as its name suggests, is a cache of pages in RAM. The pages originate from reads and writes of regular filesystem files, block device files, and memory-mapped files. In this manner, the page cache contains chunks of recently accessed files. During a page I/O operation, such as `read()`,[2] the kernel checks whether the data resides in the page cache. If the data is in the page cache, the kernel can quickly return the requested page from memory rather than read the data off the comparatively slow disk. In the rest of this chapter, we explore the data structures and kernel facilities that maintain Linux's page cache.

The `address_space` Object

A page in the page cache can consist of multiple noncontiguous physical disk blocks.[3] Checking the page cache to see whether certain data has been cached is made difficult because of this noncontiguous layout of the blocks that constitute each page. Therefore, it is not possible to index the data in the page cache using only a device name and block number, which would otherwise be the simplest solution.

Furthermore, the Linux page cache is quite general in what pages it can cache. Indeed, the original page cache introduced in System V Release 4 cached only filesystem data. Consequently, the SVR4 page cache used its equivalent of the inode object, called `struct vnode`, to manage the page cache. The Linux page cache aims to cache *any* page-based object, which includes many forms of files and memory mappings.

Although the Linux page cache could work by extending the `inode` structure (discussed in Chapter 13, "The Virtual Filesystem") to support page I/O operations, such a

---

[2] *As you saw in Chapter 13, "The Virtual Filesystem," it is not the* `read()` and `write()` *system calls that perform the actual page I/O operation, but the filesystem-specific methods specified by* `file->f_op->read()` *and* `file->f_op->write()`.

[3] *For example, a physical page is 4KB in size on the x86 architecture, whereas a disk block on many filesystems can be as small as 512 bytes. Therefore, 8 blocks might fit in a single page. The blocks need not be contiguous because the files might be laid out all over the disk.*

choice would confine the page cache to files. To maintain a generic page cache—one not tied to physical files or the inode structure—the Linux page cache uses a new object to manage entries in the cache and page I/O operations. That object is the address_space structure. Think of address_space as the physical analogue to the virtual vm_area_struct introduced in Chapter 15, "The Process Address Space." While a single file may be represented by 10 vm_area_struct structures (if, say, five processes each mmap() it twice), the file has only one address_space structure—just as the file may have many virtual addresses but exist only once in physical memory. Like much else in the Linux kernel, address_space is misnamed. A better name is perhaps page_cache_entity or physical_pages_of_a_file.

The address_space structure is defined in <linux/fs.h>:

```
struct address_space {
        struct inode            *host;              /* owning inode */
        struct radix_tree_root  page_tree;          /* radix tree of all pages */
        spinlock_t              tree_lock;          /* page_tree lock */
        unsigned int            i_mmap_writable;    /* VM_SHARED ma count */
        struct prio_tree_root   i_mmap;             /* list of all mappings */
        struct list_head        i_mmap_nonlinear;   /* VM_NONLINEAR ma list */
        spinlock_t              i_mmap_lock;        /* i_mmap lock */
        atomic_t                truncate_count;     /* truncate re count */
        unsigned long           nrpages;            /* total number of pages */
        pgoff_t                 writeback_index;    /* writeback start offset */
        struct address_space_operations   *a_ops;   /* operations table */
        unsigned long           flags;              /* gfp_mask and error flags */
        struct backing_dev_info *backing_dev_info;  /* read-ahead information */
        spinlock_t              private_lock;       /* private lock */
        struct list_head        private_list;       /* private list */
        struct address_space    *assoc_mapping;     /* associated buffers */
};
```

The i_mmap field is a priority search tree of all shared and private mappings in this address space. A priority search tree is a clever mix of heaps and radix trees.[4] Recall from earlier that while a cached file is associated with one address_space structure, it can have many vm_area_struct structures—a one-to-many mapping from the physical pages to many virtual pages. The i_mmap field allows the kernel to efficiently find the mappings associated with this cached file.

There are a total of nrpages in the address space.

The address_space is associated with some kernel object. Normally, this is an inode. If so, the host field points to the associated inode. The host field is NULL if the associated

[4] *The kernel implementation is based on the radix priority search tree proposed by Edward M. McCreight in SIAM Journal of Computing, volume 14, number 2, pages 257–276, May 1985.*

object is not an inode—for example, if the `address_space` is associated with the swapper.

`address_space` Operations

The `a_ops` field points to the address space operations table, in the same manner as the VFS objects and their operations tables. The operations table is represented by `struct address_space_operations` and is also defined in `<linux/fs.h>`:

```
struct address_space_operations {
        int (*writepage)(struct page *, struct writeback_control *);
        int (*readpage) (struct file *, struct page *);
        int (*sync_page) (struct page *);
        int (*writepages) (struct address_space *,
                           struct writeback_control *);
        int (*set_page_dirty) (struct page *);
        int (*readpages) (struct file *, struct address_space *,
                          struct list_head *, unsigned);
        int (*write_begin)(struct file *, struct address_space *mapping,
                           loff_t pos, unsigned len, unsigned flags,
                           struct page **pagep, void **fsdata);
        int (*write_end)(struct file *, struct address_space *mapping,
                         loff_t pos, unsigned len, unsigned copied,
                         struct page *page, void *fsdata);
        sector_t (*bmap) (struct address_space *, sector_t);
        int (*invalidatepage) (struct page *, unsigned long);
        int (*releasepage) (struct page *, int);
        int (*direct_IO) (int, struct kiocb *, const struct iovec *,
                          loff_t, unsigned long);
        int (*get_xip_mem) (struct address_space *, pgoff_t, int,
                            void **, unsigned long *);
        int (*migratepage) (struct address_space *,
                            struct page *, struct page *);
        int (*launder_page) (struct page *);
        int (*is_partially_uptodate) (struct page *,
                                      read_descriptor_t *,
                                      unsigned long);
        int (*error_remove_page) (struct address_space *,
                                  struct page *);
};
```

These function pointers point at the functions that implement page I/O for this cached object. Each backing store describes how it interacts with the page cache via its own `address_space_operations`. For example, the ext3 filesystem defines its operations in `fs/ext3/inode.c`. Thus, these are the functions that manage the page cache, including the most common: reading pages into the cache and updated data in the cache. Thus, the `readpage()` and `writepage()` methods are most important. Let's look at the steps

involved in each, starting with a page read operation. First, the Linux kernel attempts to find the request data in the page cache. The `find_get_page()` method is used to perform this check; it is passed an `address_space` and page offset. These values search the page cache for the desired data:

```
page = find_get_page(mapping, index);
```

Here, `mapping` is the given `address_space` and `index` is the desired offset into the file, in pages. (Yes, calling the `address_space` structure `mapping` just furthers the naming confusion. I'm replicating the kernel's naming for consistency, but I do not condone it.) If the page does not exist in the cache, `find_get_page()` returns `NULL` and a new page is allocated and added to the page cache:

```
struct page *page;
int error;

/* allocate the page ... */
page = page_cache_alloc_cold(mapping);
if (!page)
        /* error allocating memory */

/* ... and then add it to the page cache */
error = add_to_page_cache_lru(page, mapping, index, GFP_KERNEL);
if (error)
        /* error adding page to page cache */
```

Finally, the requested data can be read from disk, added to the page cache, and returned to the user:

```
error = mapping->a_ops->readpage(file, page);
```

Write operations are a bit different. For file mappings, whenever a page is modified, the VM simply calls

```
SetPageDirty(page);
```

The kernel later writes the page out via the `writepage()` method. Write operations on specific files are more complicated. The generic write path in `mm/filemap.c` performs the following steps:

```
page = __grab_cache_page(mapping, index, &cached_page, &lru_pvec);
status = a_ops->prepare_write(file, page, offset, offset+bytes);
page_fault = filemap_copy_from_user(page, offset, buf, bytes);
status = a_ops->commit_write(file, page, offset, offset+bytes);
```

First, the page cache is searched for the desired page. If it is not in the cache, an entry is allocated and added. Next, the kernel sets up the write request and the data is copied from user-space into a kernel buffer. Finally, the data is written to disk.

Because the previous steps are performed during all page I/O operations, all page I/O is guaranteed to go through the page cache. Consequently, the kernel attempts to satisfy

all read requests from the page cache. If this fails, the page is read in from disk and added to the page cache. For write operations, the page cache acts as a staging ground for the writes. Therefore, all written pages are also added to the page cache.

Radix Tree

Because the kernel must check for the existence of a page in the page cache before initiating any page I/O, such a check must be quick. Otherwise, the overhead of searching and checking the page cache could nullify any benefits the cache might provide. (At least if the cache hit rate is low—the overhead would have to be awful to cancel out the benefit of retrieving the data from memory in lieu of disk.)

As you saw in the previous section, the page cache is searched via the `address_space` object plus an offset value. Each `address_space` has a unique radix tree stored as `page_tree`. A radix tree is a type of binary tree. The radix tree enables quick searching for the desired page, given only the file offset. Page cache searching functions such as `find_get_page()` call `radix_tree_lookup()`, which performs a search on the given tree for the given object.

The core radix tree code is available in generic form in `lib/radix-tree.c`. Users of the radix tree need to include `<linux/radix-tree.h>`.

The Old Page Hash Table

Prior to the 2.6 kernel, the page cache was not searched via the radix tree. Instead, a global hash was maintained over all the pages in the system. The hash returned a doubly linked list of entries that hash to the same given value. If the desired page were in the cache, one of the items in the list was the corresponding page. Otherwise, the page was not in the page cache and the hash function returned `NULL`.

The global hash had four primary problems:

- A single global lock protected the hash. Lock contention was quite high on even moderately sized machines, and performance suffered as a result.
- The hash was larger than necessary because it contained all the pages in the page cache, whereas only pages pertaining to the current file were relevant.
- Performance when the hash lookup failed (that is, the given page was not in the page cache) was slower than desired, particularly because it was necessary to walk the chains off of a given hash value.
- The hash consumed more memory than other possible solutions.

The introduction of the radix tree-based page cache in 2.6 solved these issues.

The Buffer Cache

Individual disk blocks also tie into the page cache, by way of block I/O buffers. Recall from Chapter 14, "The Block I/O Layer," that a buffer is the in-memory representation of a single physical disk block. Buffers act as descriptors that map pages in memory to

disk blocks; thus, the page cache also reduces disk access during block I/O operations by both caching disk blocks and buffering block I/O operations until later. This caching is often referred to as the *buffer cache*, although as implemented it is not a separate cache but is part of the page cache.

Block I/O operations manipulate a single disk block at a time. A common block I/O operation is reading and writing inodes. The kernel provides the `bread()` function to perform a low-level read of a single block from disk. Via buffers, disk blocks are mapped to their associated in-memory pages and cached in the page cache.

The buffer and page caches were not always unified; doing so was a major feature of the 2.4 Linux kernel. In earlier kernels, there were two separate disk caches: the page cache and the buffer cache. The former cached pages; the latter cached buffers. The two caches were not unified: A disk block could exist in both caches simultaneously. This led to wasted effort synchronizing the two cached copies and memory wasted in duplicating cached items. Today, we have one disk cache: the page cache. The kernel still needs to use buffers, however, to represent disk blocks in memory. Conveniently, the buffers describe the mapping of a block onto a page, which is in the page cache.

The Flusher Threads

Write operations are deferred in the page cache. When data in the page cache is newer than the data on the backing store, we call that data *dirty*. Dirty pages that accumulate in memory eventually need to be written back to disk. Dirty page writeback occurs in three situations:

- When free memory shrinks below a specified threshold, the kernel writes dirty data back to disk to free memory because only clean (nondirty) memory is available for eviction. When clean, the kernel can evict the data from the cache and then shrink the cache, freeing up more memory.
- When dirty data grows older than a specific threshold, sufficiently old data is written back to disk to ensure that dirty data does not remain dirty indefinitely.
- When a user process invokes the `sync()` and `fsync()` system calls, the kernel performs writeback on demand.

These three jobs have rather different goals. In fact, two separate kernel threads performed the work in older kernels (see the following section). In 2.6, however, a gang[5] of kernel threads, the *flusher threads*, performs all three jobs.

First, the flusher threads need to flush dirty data back to disk when the amount of free memory in the system shrinks below a specified level. The goal of this background writeback is to regain memory consumed by dirty pages when available physical memory is

[5] *The term "gang" is commonly used in computer science to denote a group of things that can operate in parallel.*

low. The memory level at which this process begins is configured by the `dirty_background_ratio` sysctl. When free memory drops below this threshold, the kernel invokes the `wakeup_flusher_threads()` call to wake up one or more flusher threads and have them run the `bdi_writeback_all ()` function to begin writeback of dirty pages. This function takes as a parameter the number of pages to attempt to write back. The function continues writing out data until two conditions are true:

- The specified minimum number of pages has been written out.
- The amount of free memory is above the `dirty_background_ratio` threshold.

These conditions ensure that the flusher threads do their part to relieve low-memory conditions. Writeback stops prior to these conditions only if the flusher threads write back *all* the dirty pages and there is nothing left to do.

For its second goal, a flusher thread periodically wakes up (unrelated to low-memory conditions) and writes out old dirty pages. This is performed to ensure that no dirty pages remain in memory indefinitely. During a system failure, because memory is volatile, dirty pages in memory that have not been written to disk are lost. Consequently, periodically synchronizing the page cache with the disk is important. On system boot, a timer is initialized to wake up a flusher thread and have it run the `wb_writeback()` function. This function then writes back all data that was modified longer than `dirty_expire_interval` milliseconds ago. The timer is then reinitialized to expire again in `dirty_writeback_interval` milliseconds. In this manner, the flusher threads periodically wake up and write to disk all dirty pages older than a specified limit.

The system administrator can set these values either in `/proc/sys/vm` or via sysctl. Table 16.1 lists the variables.

Table 16.1 **Page Writeback Settings**

Variable	Description
`dirty_background_ratio`	As a percentage of total memory, the number of pages at which the flusher threads begin writeback of dirty data.
`dirty_expire_interval`	In milliseconds, how old data must be to be written out the next time a flusher thread wakes to perform periodic writeback.
`dirty_ratio`	As a percentage of total memory, the number of pages a process generates before it begins writeback of dirty data.
`dirty_writeback_interval`	In milliseconds, how often a flusher thread should wake up to write data back out to disk.
`laptop_mode`	A Boolean value controlling *laptop mode*. See the following section.

The flusher code lives in `mm/page-writeback.c` and `mm/backing-dev.c` and the writeback mechanism lives in `fs/fs-writeback.c`.

Laptop Mode

Laptop mode is a special page writeback strategy intended to optimize battery life by minimizing hard disk activity and enabling hard drives to remain spun down as long as possible. It is configurable via `/proc/sys/vm/laptop_mode`. By default, this file contains a zero and laptop mode is disabled. Writing a one to this file enables laptop mode.

Laptop mode makes a single change to page writeback behavior. In addition to performing writeback of dirty pages when they grow too old, the flusher threads also piggyback off any other physical disk I/O, flushing *all* dirty buffers to disk. In this manner, page writeback takes advantage that the disk was just spun up, ensuring that it will not cause the disk to spin up later.

This behavioral change makes the most sense when `dirty_expire_interval` and `dirty_writeback_interval` are set to large values—say, 10 minutes. With writeback so delayed, the disk is spun up infrequently, and when it does spin up, laptop mode ensures that the opportunity is well utilized. Because shutting off the disk drive is a significant source of power savings, laptop mode can greatly improve how long a laptop lasts on battery. The downside is that a system crash or other failure can lose a lot of data.

Many Linux distributions automatically enable and disable laptop mode, and modify other writeback tunables, when going on and off battery. This enables a machine to benefit from laptop mode when on battery power and then automatically return to normal page writeback behavior when plugged into AC.

History: bdflush, kupdated, and pdflush

Prior to the 2.6 kernel, the job of the flusher threads was met by two other kernel threads, *bdflush* and *kupdated*.

The bdflush kernel thread performed background writeback of dirty pages when available memory was low. A set of thresholds was maintained, similar to the flusher threads', and bdflush was awakened via `wakeup_bdflush()` whenever free memory dropped below those thresholds.

Two main differences distinguish bdflush and the current flusher threads. The first, which is discussed in the next section, is that there was always only one bdflush daemon, whereas the number of flusher threads is a function of the number of disk spindles. The second difference is that bdflush was buffer-based; it wrote back dirty buffers. Conversely, the flusher threads are page-based; they write back whole pages. Of course, the pages can correspond to buffers, but the actual I/O unit is a full page and not a single buffer. This is beneficial as managing pages is easier than managing buffers because pages are a more general and common unit.

Because bdflush flushes buffers only when memory is low or the number of buffers is too large, the kupdated thread was introduced to periodically write back dirty pages. It served an identical purpose to the `wb_writeback()` function.

In the 2.6 kernel, bdflush and kupdated gave way to the *pdflush threads*. Short for *page dirty flush* (more of those confusing names), the pdflush threads performed similar to the flusher threads of today. The main difference is that the number of pdflush threads is dynamic, by default between two and eight, depending on the I/O load of the system. The pdflush threads are not associated with any specific disk; instead, they are global to all disks in the system. This allows for a simple implementation. The downside is that pdflush can easily trip up on congested disks, and congestion is easy to cause with modern hardware. Moving to per-spindle flushing enables the I/O to perform synchronously, simplifying the congestion logic and improving performance. The flusher threads replaced the pdflush threads in the 2.6.32 kernel. The per-spindle flushing is the main difference; the rest of this section is also applicable to pdflush and thus any 2.6 kernel.

Avoiding Congestion with Multiple Threads

One of the major flaws in the bdflush solution was that bdflush consisted of one thread. This led to possible congestion during heavy page writeback where the single bdflush thread would block on a single congested device queue (the list of I/O requests waiting to submit to disk), whereas other device queues would sit relatively idle. If the system has multiple disks and the associated processing power, the kernel should keep each disk busy. Unfortunately, even with plenty of data needing writeback, bdflush can become stuck handling a single queue and fail to keep all disks saturated. This occurs because the throughput of disks is a finite—and unfortunately comparatively small—number. If only a single thread is performing page writeback, that single thread can easily spend a long time waiting for a single disk because disk throughput is such a limiting quantity. To mitigate this, the kernel needs to multithread page writeback. In this manner, no single device queue can become a bottleneck.

The 2.6 kernel solves this problem by enabling multiple flusher threads to exist. Each thread individually flushes dirty pages to disk, allowing different flusher threads to concentrate on different device queues. With the pdflush threads, the number of threads was dynamic, and each thread tried to stay busy grabbing data from the per-superblock dirty list and writing it back to disk. The pdflush approach prevents a single busy disk from starving other disks. This is all good, but what if each pdflush thread were to get hung up writing to the same, congested, queue? In that case, the performance of multiple pdflush threads would not be an improvement over a single thread. The memory consumed, however, would be significantly greater. To mitigate this effect, the pdflush threads employ congestion avoidance: They actively try to write back pages whose queues are not congested. As a result, the pdflush threads spread out their work and refrain from merely hammering on the same busy device.

This approach worked fairly well, but the congestion avoidance was not perfect. On modern systems, congestion is easy to cause because I/O bus technology improves at a slower rate than the rest of the computer—processors keep getting faster according to Moore's Law, but hard drives are only marginally quicker than they were two decades ago. Moreover, aside from pdflush, no other part of the I/O system employs congestion

avoidance. Thus, in certain cases pdflush can avoid writing back on a specific disk far longer than desired. With the current flusher threads model, available since 2.6.32, the threads are associated with a block device, so each thread grabs data from its per-block device dirty list and writes it back to its disk. Writeback is thus synchronous and the threads, because there is one per disk, do not need to employ complicated congestion avoidance. This approach improves fairness and decreases the risk of starvation.

Because of the improvements in page writeback, starting with the introduction of pdflush and continuing with the flusher threads, the 2.6 kernel can keep many more disks saturated than any earlier kernel. In the face of heavy activity, the flusher threads can maintain high throughput across multiple disks.

Conclusion

This chapter looked at Linux's page cache and page writeback. We saw how the kernel performs all page I/O through the page cache and how this page cache, by storing data in memory, significantly improves the performance of the system by reducing the amount of disk I/O. We discussed how writes are maintained in the page cache through a process called write-back caching, which keeps pages "dirty" in memory and defers writing the data back to disk. The flusher "gang" of kernel threads handles this eventual page writeback.

Over the last few chapters, we have built a solid understanding of memory and filesystem management. Now let's segue over to the topic of device drivers and modules to see how the Linux kernel provides a modular and dynamic infrastructure for the run-time insertion and removal of kernel code.