Chapter 6

Buffer cache

One of an operating system's central roles is to enable safe cooperation between processes sharing a computer. First, it must isolate the processes from each other, so that one errant process cannot harm the operation of others. To do this, xv6 uses the x86 hardware's memory segmentation (Chapter 2). Second, an operating system must provide controlled mechanisms by which the now-isolated processes can overcome the isolation and cooperate. To do this, xv6 provides the concept of files. One process can write data to a file, and then another can read it; processes can also be more tightly coupled using pipes. The next four chapters examine the implementation of files, working up from individual disk blocks to disk data structures to directories to system calls. This chapter examines the disk driver and the buffer cache, which together form the bottom layer of the file implementation.

The disk driver copies data from and back to the disk, The buffer cache manages these temporary copies of the disk blocks. Caching disk blocks has an obvious performance benefit: disk access is significantly slower than memory access, so keeping frequently-accessed disk blocks in memory reduces the number of disk accesses and makes the system faster. Even so, performance is not the most important reason for the buffer cache. When two different processes need to edit the same disk block (for example, perhaps both are creating files in the same directory), the disk block is shared data, just like the process table is shared among all kernel threads in Chapter 5. The buffer cache serializes access to the disk blocks, just as locks serialize access to in-memory data structures. Like the operating system as a whole, the buffer cache's fundamental purpose is to enable safe cooperation between processes.

Code: Data structures

Disk hardware traditionally presents the data on the disk as a numbered sequence of 512-byte blocks called sectors: sector 0 is the first 512 bytes, sector 1 is the next, and so on. The disk drive and buffer cache coordinate the use of disk sectors with a data structure called a buffer, `struct buf` (3400). Each buffer represents the contents of one sector on a particular disk device. The `dev` and `sector` fields give the device and sector number and the `data` field is an in-memory copy of the disk sector. The `data` is often out of sync with the disk: it might have not yet been read in from disk, or it might have been updated but not yet written out. The `flags` track the relationship between memory and disk: the `B_VALID` flag means that `data` has been read in, and the B_DIRTY flag means that `data` needs to be written out. The `B_BUSY` flag is a lock bit; it indicates that some process is using the buffer and other processes must not.

When a buffer has the B_BUSY flag set, we say the buffer is locked.

Code: Disk driver

The IDE device provides access to disks connected to the PC standard IDE controller. IDE is now falling out of fashion in favor of SCSI and SATA, but the interface is very simple and lets us concentrate on the overall structure of a driver instead of the details of a particular piece of hardware.

The kernel initializes the disk driver at boot time by calling ideinit (3751) from main (1360). Ideinit initializes idelock (3722) and then must prepare the hardware. In Chapter 3, xv6 disabled all hardware interrupts. Ideinit calls picenable and ioapicenable to enable the IDE_IRQ interrupt (3756-3757). The call to picenable enables the interrupt on a uniprocessor; ioapicenable enables the interrupt on a multiprocessor, but only on the last CPU (ncpu-1): on a two-processor system, CPU 1 handles disk interrupts.

Next, ideinit probes the disk hardware. It begins by calling idewait (3758) to wait for the disk to be able to accept commands. The disk hardware presents status bits on port 0x1f7, as we saw in chapter 1. Idewait (3732) polls the status bits until the busy bit (IDE_BSY) is clear and the ready bit (IDE_DRDY) is set.

Now that the disk controller is ready, ideinit can check how many disks are present. It assumes that disk 0 is present, because the boot loader and the kernel were both loaded from disk 0, but it must check for disk 1. It writes to port 0x1f6 to select disk 1 and then waits a while for the status bit to show that the disk is ready (3760-3767). If not, ideinit assumes the disk is absent.

After ideinit, the disk is not used again until the buffer cache calls iderw, which updates a locked buffer as indicated by the flags. If B_DIRTY is set, iderw writes the buffer to the disk; if B_VALID is not set, iderw reads the buffer from the disk.

Disk accesses typically take milliseconds, a long time for a processor. In Chapter 1, the boot sector issues disk read commands and reads the status bits repeatedly until the data is ready. This polling or busy waiting is fine in a boot sector, which has nothing better to do. In an operating system, however, it is more efficient to let another process run on the CPU and arrange to receive an interrupt when the disk operation has completed. Iderw takes this latter approach, keeping the list of pending disk requests in a queue and using interrupts to find out when each request has finished. Although iderw maintains a queue of requests, the simple IDE disk controller can only handle one operation at a time. The disk driver maintains the invariant that it has sent the buffer at the front of the queue to the disk hardware; the others are simply waiting their turn.

Iderw (3854) adds the buffer b to the end of the queue (3867-3871). If the buffer is at the front of the queue, iderw must send it to the disk hardware by calling idestart (3824-3826); otherwise the buffer will be started once the buffers ahead of it are taken care of.

Idestart (3775) issues either a read or a write for the buffer's device and sector, according to the flags. If the operation is a write, idestart must supply the data now

(3789) and the interrupt will signal that the data has been written to disk. If the operation is a read, the interrupt will signal that the data is ready, and the handler will read it.

Having added the request to the queue and started it if necessary, `iderw` must wait for the result. As discussed above, polling does not make efficient use of the CPU. Instead, `iderw` sleeps, waiting for the interrupt handler to record in the buffer's flags that the operation is done (3879-3880). While this process is sleeping, xv6 will schedule other processes to keep the CPU busy.

Eventually, the disk will finish its operation and trigger an interrupt. As we saw in Chapter 3, `trap` will call `ideintr` to handle it (3024). `Ideintr` (3802) consults the first buffer in the queue to find out which operation was happening. If the buffer was being read and the disk controller has data waiting, `ideintr` reads the data into the buffer with `insl` (3815-3817). Now the buffer is ready: `ideintr` sets `B_VALID`, clears `B_DIRTY`, and wakes up any process sleeping on the buffer (3819-3822). Finally, `ideintr` must pass the next waiting buffer to the disk (3824-3826).

Code: Interrupts and locks

On a multiprocessor, ordinary kernel code can run on one CPU while an interrupt handler runs on another. If the two code sections share data, they must use locks to synchronize access to that data. For example, `iderw` and `ideintr` share the request queue and use `idelock` to synchronize.

Interrupts can cause concurrency even on a single processor: if interrupts are enabled, kernel code can be stopped at any moment to run an interrupt handler instead. Suppose `iderw` held the `idelock` and then got interrupted to run `ideintr. Ideintr` would try to lock `idelock`, see it was held, and wait for it to be released. In this situation, `idelock` will never be released—only `iderw` can release it, and `iderw` will not continue running until `ideintr` returns—so the processor, and eventually the whole system, will deadlock. To avoid this situation, if a lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled. Xv6 is more conservative: it never holds any lock with interrupts enabled. It uses `pushcli` (1605) and `popcli` (1616) to manage a stack of “disable interrupts” operations (`cli` is the x86 instruction that disables interrupts, as we saw in Chapter 1). `Acquire` calls `pushcli` before trying to acquire a lock (1525), and `release` calls `popcli` after releasing the lock (1571). It is important that `acquire` call `pushcli` before the `xchg` that might acquire the lock (1532). If the two were reversed, there would be a few instruction cycles when the lock was held with interrupts enabled, and an unfortunately timed interrupt would deadlock the system. Similarly, it is important that `release` call `popcli` only after the `xchg` that releases the lock (1532). These races are similar to the ones involving `holding` (see Chapter 4).

Code: Buffer cache

As discussed at the beginning of this chapter, the buffer cache synchronizes access to disk blocks, making sure that only one kernel process at a time can edit the file system data in any particular buffer. The buffer cache does this by blocking processes in `bread` (pronounced b-read): if two processes call `bread` with the same device and sector number of an otherwise unused disk block, the call in one process will return a buffer immediately; the call in the other process will not return until the first process has signaled that it is done with the buffer by calling `brelse` (b-release).

The buffer cache is a doubly-linked list of buffers. `Binit`, called by `main` (1357), initializes the list with the `NBUF` buffers in the static array `buf` (3950-3959). All other access to the buffer cache refer to the linked list via `bcache.head`, not the `buf` array.

`Bread` (4002) calls `bget` to get a locked buffer for the given sector (4006). If the buffer needs to be read from disk, `bread` calls `iderw` to do that before returning the buffer.

`Bget` (3966) scans the buffer list for a buffer with the given device and sector numbers (3973-3984). If there is such a buffer, `bget` needs to lock it before returning. If the buffer is not in use, `bget` can set the `B_BUSY` flag and return (3976-3983). If the buffer is already in use, `bget` sleeps on the buffer to wait for its release. When `sleep` returns, `bget` cannot assume that the buffer is now available. In fact, since `sleep` released and reacquired `buf_table_lock`, there is no guarantee that `b` is still the right buffer: maybe it has been reused for a different disk sector. `Bget` has no choice but to start over (3982), hoping that the outcome will be different this time.

If there is no buffer for the given sector, `bget` must make one, possibly reusing a buffer that held a different sector. It scans the buffer list a second time, looking for a block that is not busy: any such block can be used (3986-3988). `Bget` edits the block metadata to record the new device and sector number and mark the block busy before returning the block (3991-3993). Note that the assignment to `flags` not only sets the `B_BUSY` bit but also clears the `B_VALID` and `B_DIRTY` bits, making sure that `bread` will refresh the buffer data from disk rather than use the previous block's contents.

Because the buffer cache is used for synchronization, it is important that there is only ever one buffer for a particular disk sector. The assignments (3989-3991) are only safe because `bget`'s first loop determined that no buffer already existed for that sector, and `bget` has not given up `buf_table_lock` since then.

If all the buffers are busy, something has gone wrong: `bget` panics. A more graceful response would be to sleep until a buffer became free, though there would be a possibility of deadlock.

Once `bread` has returned a buffer to its caller, the caller has exclusive use of the buffer and can read or write the data bytes. If the caller does write to the data, it must call `bwrite` to flush the changed data out to disk before releasing the buffer. `Bwrite` (4014) sets the `B_DIRTY` flag and calls `iderw` to write the buffer to disk.

When the caller is done with a buffer, it must call `brelse` to release it. (The name `brelse`, a shortening of b-release, is cryptic but worth learning: it originated in Unix and is used in BSD, Linux, and Solaris too.) `Brelse` (4024) moves the buffer from its position in the linked list to the front of the list (4031-4036), clears the `B_BUSY` bit, and wakes any processes sleeping on the buffer. Moving the buffer has the effect that the buffers are ordered by how recently they were used (meaning released): the first buffer in the list is the most recently used, and the last is the least recently used. The two loops in bget take advantage of this: the scan for an existing buffer must process the entire list in the worst case, but checking the most recently used buffers first (starting at bcache.head and following next pointers) will reduce scan time when there is good locality of reference. The scan to pick a buffer to reuse picks the least recently used block by scanning backward (following prev pointers); the implicit assumption is that the least recently used buffer is the one least likely to be used again soon.

Real world

Actual device drivers are far more complex than the disk driver in this chapter, but the basic ideas are the same: typically devices are slower than CPU, so the hardware uses interrupts to notify the operating system of status changes. Modern disk controllers typically accept multiple outstanding disk requests at a time and even reorder them to make most efficient use of the disk arm. When disks were simpler, operating system often reordered the request queue themselves, though reordering has implications for file system consistency, as we will see in Chapter 8.

Other hardware is surprisingly similar to disks: network device buffers hold packets, audio device buffers hold sound samples, graphics card buffers hold video data and command sequences. High-bandwidth devices—disks, graphics cards, and network cards—often use direct memory access (DMA) instead of the explicit i/o (insl, outsl) in this driver. DMA allows the disk or other controllers direct access to physical memory. The driver gives the device the physical address of the buffer's data field and the device copies directly to or from main memory, interrupting once the copy is complete. Using DMA means that the CPU is not involved at all in the transfer, which can be more efficient and is less taxing for the CPU's memory caches.

The buffer cache in a real-world operating system is significantly more complex than xv6's, but it serves the same two purposes: caching and synchronizing access to the disk. Xv6's buffer cache, like V6's, uses a simple least recently used (LRU) eviction policy; there are many more complex policies that can be implemented, each good for some workloads and not as good for others. A more efficient LRU cache would eliminate the linked list, instead using a hash table for lookups and a heap for LRU evictions.

In real-world operating systems, buffers typically match the hardware page size, so that read-only copies can be mapped into a process's address space using the paging hardware, without any copying.

Exercises

1. Setting a bit in a buffer's flags is not an atomic operation: the processor makes a copy of flags in a register, edits the register, and writes it back. Thus it is important that two processes are not writing to flags at the same time. The code in this chapter edits the B_BUSY bit only while holding buflock but edits the B_VALID and B_WRITE flags without holding any locks. Why is this safe?