36

I/O Devices

Before delving into the main content of this part of the book (on persistence), we first introduce the concept of an **input/output (I/O) device** and show how the operating system might interact with such an entity. I/O is quite critical to computer systems, of course; imagine a program without any input (it produces the same result each time); now imagine a program with no output (what was the purpose of it running?). Clearly, for computer systems to be interesting, both input and output are required. And thus, our general problem:

CRUX: HOW TO INTEGRATE I/O INTO SYSTEMS
How should I/O be integrated into systems? What are the general mechanisms? How can we make them efficient?

36.1 System Architecture

To begin our discussion, let's look at a "classical" diagram of a typical system (Figure 36.1, page 2). The picture shows a single CPU attached to the main memory of the system via some kind of **memory bus** or interconnect. Some devices are connected to the system via a general **I/O bus**, which in many modern systems would be **PCI** (or one of its many derivatives); graphics and some other higher-performance I/O devices might be found here. Finally, even lower down are one or more of what we call a **peripheral bus**, such as **SCSI**, **SATA**, or **USB**. These connect slow devices to the system, including **disks**, **mice**, and **keyboards**.

One question you might ask is: why do we need a hierarchical structure like this? Put simply: physics, and cost. The faster a bus is, the shorter it must be; thus, a high-performance memory bus does not have much room to plug devices and such into it. In addition, engineering a bus for high performance is quite costly. Thus, system designers have adopted this hierarchical approach, where components that demand high performance (such as the graphics card) are nearer the CPU. Lower performance components are further away. The benefits of placing disks and other slow devices on a peripheral bus are manifold; in particular, you can place a large number of devices on it.

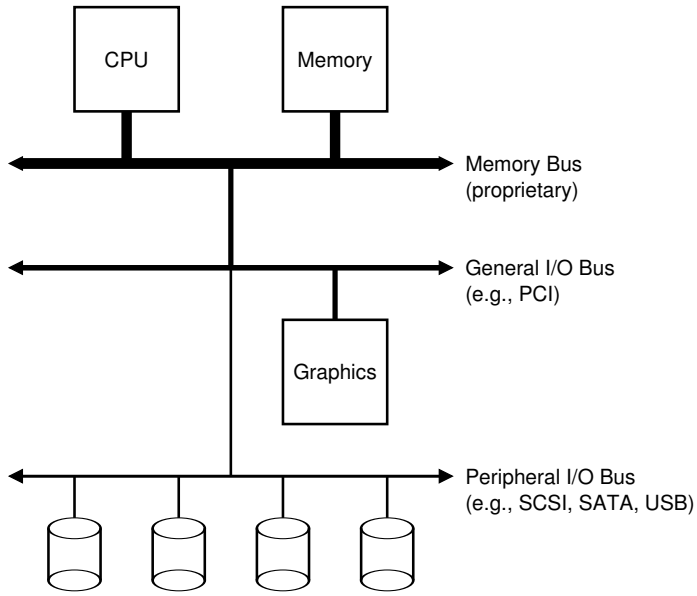


Figure 36.1: **Prototypical System Architecture**

Of course, modern systems increasingly use specialized chipsets and faster point-to-point interconnects to improve performance. Figure 36.2 (page 3) shows an approximate diagram of Intel's Z270 Chipset [H17]. Along the top, the CPU connects most closely to the memory system, but also has a high-performance connection to the graphics card (and thus, the display) to enable gaming (oh, the horror!) and other graphics-intensive applications.

The CPU connects to an I/O chip via Intel's proprietary **DMI** (**Direct Media Interface**), and the rest of the devices connect to this chip via a number of different interconnects. On the right, one or more hard drives connect to the system via the **eSATA** interface; **ATA** (the **AT Attachment**, in reference to providing connection to the IBM PC AT), then **SATA** (for **Serial ATA**), and now eSATA (for **external SATA**) represent an evolution of storage interfaces over the past decades, with each step forward increasing performance to keep pace with modern storage devices.

Below the I/O chip are a number of **USB** (**Universal Serial Bus**) connections, which in this depiction enable a keyboard and mouse to be attached to the computer. On many modern systems, USB is used for low performance devices such as these.

Finally, on the left, other higher performance devices can be connected

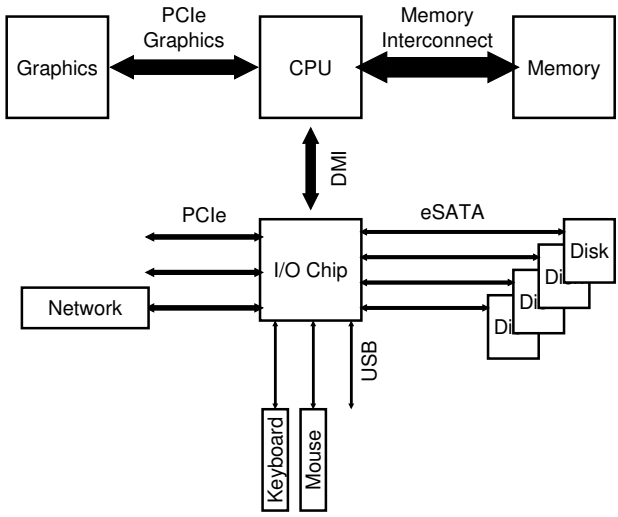


Figure 36.2: **Modern System Architecture**

to the system via **PCIe** (**Peripheral Component Interconnect Express**). In this diagram, a network interface is attached to the system here; higher performance storage devices (such as **NVMe** persistent storage devices) are often connected here.

36.2 A Canonical Device

Let us now look at a canonical device (not a real one), and use this device to drive our understanding of some of the machinery required to make device interaction efficient. From Figure 36.3 (page 4), we can see that a device has two important components. The first is the hardware **interface** it presents to the rest of the system. Just like a piece of software, hardware must also present some kind of interface that allows the system software to control its operation. Thus, all devices have some specified interface and protocol for typical interaction.

The second part of any device is its **internal structure**. This part of the device is implementation specific and is responsible for implementing the abstraction the device presents to the system. Very simple devices will have one or a few hardware chips to implement their functionality; more complex devices will include a simple CPU, some general purpose memory, and other device-specific chips to get their job done. For example, modern RAID controllers might consist of hundreds of thousands of lines of **firmware** (i.e., software within a hardware device) to implement its functionality.

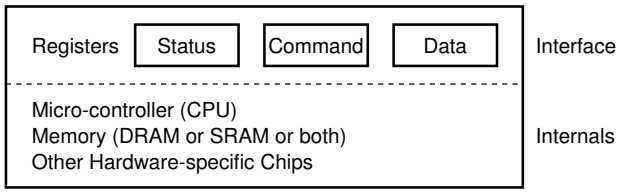


Figure 36.3: **A Canonical Device**

36.3 The Canonical Protocol

In the picture above, the (simplified) device interface is comprised of three registers: a **status** register, which can be read to see the current status of the device; a **command** register, to tell the device to perform a certain task; and a **data** register to pass data to the device, or get data from the device. By reading and writing these registers, the operating system can control device behavior.

Let us now describe a typical interaction that the OS might have with the device in order to get the device to do something on its behalf. The protocol is as follows:

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

The protocol has four steps. In the first, the OS waits until the device is ready to receive a command by repeatedly reading the status register; we call this **polling** the device (basically, just asking it what is going on). Second, the OS sends some data down to the data register; one can imagine that if this were a disk, for example, that multiple writes would need to take place to transfer a disk block (say 4KB) to the device. When the main CPU is involved with the data movement (as in this example protocol), we refer to it as **programmed I/O (PIO)**. Third, the OS writes a command to the command register; doing so implicitly lets the device know that both the data is present and that it should begin working on the command. Finally, the OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished (it may then get an error code to indicate success or failure).

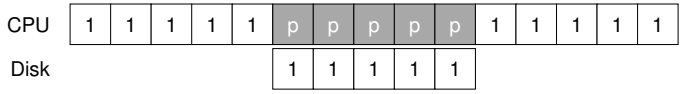
This basic protocol has the positive aspect of being simple and working. However, there are some inefficiencies and inconveniences involved. The first problem you might notice in the protocol is that polling seems inefficient; specifically, it wastes a great deal of CPU time just waiting for the (potentially slow) device to complete its activity, instead of switching to another ready process and thus better utilizing the CPU.

THE CRUX: HOW TO AVOID THE COSTS OF POLLING
How can the OS check device status without frequent polling, and thus lower the CPU overhead required to manage the device?

36.4 Lowering CPU Overhead With Interrupts

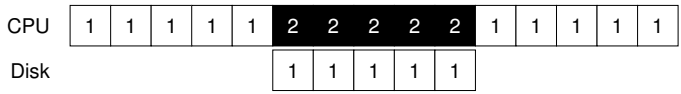
The invention that many engineers came upon years ago to improve this interaction is something we've seen already: the **interrupt**. Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task. When the device is finally finished with the operation, it will raise a hardware interrupt, causing the CPU to jump into the OS at a predetermined **interrupt service routine (ISR)** or more simply an **interrupt handler**. The handler is just a piece of operating system code that will finish the request (for example, by reading data and perhaps an error code from the device) and wake the process waiting for the I/O, which can then proceed as desired.

Interrupts thus allow for **overlap** of computation and I/O, which is key for improved utilization. This timeline shows the problem:



In the diagram, Process 1 runs on the CPU for some time (indicated by a repeated `1` on the CPU line), and then issues an I/O request to the disk to read some data. Without interrupts, the system simply spins, polling the status of the device repeatedly until the I/O is complete (indicated by a `p`). The disk services the request and finally Process 1 can run again.

If instead we utilize interrupts and allow for overlap, the OS can do something else while waiting for the disk:



In this example, the OS runs Process 2 on the CPU while the disk services Process 1's request. When the disk request is finished, an interrupt occurs, and the OS wakes up Process 1 and runs it again. Thus, *both* the CPU and the disk are properly utilized during the middle stretch of time.

Note that using interrupts is not *always* the best solution. For example, imagine a device that performs its tasks very quickly: the first poll usually finds the device to be done with task. Using an interrupt in this case will actually *slow down* the system: switching to another process, handling the interrupt, and switching back to the issuing process is expensive. Thus, if a device is fast, it may be best to poll; if it is slow, interrupts, which allow

TIP: INTERRUPTS NOT ALWAYS BETTER THAN POLLING
Although interrupts allow for overlap of computation and I/O, they only really make sense for slow devices. Otherwise, the cost of interrupt handling and context switching may outweigh the benefits interrupts provide. There are also cases where a flood of interrupts may overload a system and lead it to livelock [MR96]; in such cases, polling provides more control to the OS in its scheduling and thus is again useful.

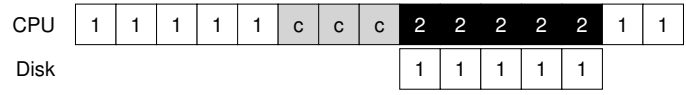
overlap, are best. If the speed of the device is not known, or sometimes fast and sometimes slow, it may be best to use a **hybrid** that polls for a little while and then, if the device is not yet finished, uses interrupts. This **two-phased** approach may achieve the best of both worlds.

Another reason not to use interrupts arises in networks [MR96]. When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to **livelock**, that is, find itself only processing interrupts and never allowing a user-level process to run and actually service the requests. For example, imagine a web server that experiences a load burst because it became the top-ranked entry on hacker news [H18]. In this case, it is better to occasionally use polling to better control what is happening in the system and allow the web server to service some requests before going back to the device to check for more packet arrivals.

Another interrupt-based optimization is **coalescing**. In such a setup, a device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU. While waiting, other requests may soon complete, and thus multiple interrupts can be coalesced into a single interrupt delivery, thus lowering the overhead of interrupt processing. Of course, waiting too long will increase the latency of a request, a common trade-off in systems. See Ahmad et al. [A+11] for an excellent summary.

36.5 More Efficient Data Movement With DMA

Unfortunately, there is one other aspect of our canonical protocol that requires our attention. In particular, when using programmed I/O (PIO) to transfer a large chunk of data to a device, the CPU is once again overburdened with a rather trivial task, and thus wastes a lot of time and effort that could better be spent running other processes. This timeline illustrates the problem:

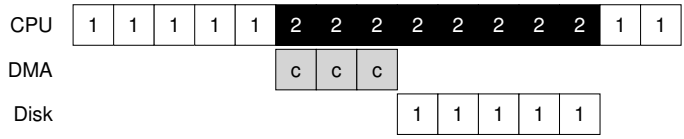


In the timeline, Process 1 is running and then wishes to write some data to the disk. It then initiates the I/O, which must copy the data from memory to the device explicitly, one word at a time (marked `c` in the diagram). When the copy is complete, the I/O begins on the disk and the CPU can finally be used for something else.

THE CRUX: HOW TO LOWER PIO OVERHEADS
With PIO, the CPU spends too much time moving data to and from devices by hand. How can we offload this work and thus allow the CPU to be more effectively utilized?

The solution to this problem is something we refer to as **Direct Memory Access (DMA)**. A DMA engine is essentially a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention.

DMA works as follows. To transfer data to the device, for example, the OS would program the DMA engine by telling it where the data lives in memory, how much data to copy, and which device to send it to. At that point, the OS is done with the transfer and can proceed with other work. When the DMA is complete, the DMA controller raises an interrupt, and the OS thus knows the transfer is complete. The revised timeline:



From the timeline, you can see that the copying of data is now handled by the DMA controller. Because the CPU is free during that time, the OS can do something else, here choosing to run Process 2. Process 2 thus gets to use more CPU before Process 1 runs again.

36.6 Methods Of Device Interaction

Now that we have some sense of the efficiency issues involved with performing I/O, there are a few other problems we need to handle to incorporate devices into modern systems. One problem you may have noticed thus far: we have not really said anything about how the OS actually communicates with the device! Thus, the problem:

THE CRUX: HOW TO COMMUNICATE WITH DEVICES
How should the hardware communicate with a device? Should there be explicit instructions? Or are there other ways to do it?

Over time, two primary methods of device communication have developed. The first, oldest method (used by IBM mainframes for many years) is to have explicit **I/O instructions**. These instructions specify a way for the OS to send data to specific device registers and thus allow the construction of the protocols described above.

For example, on x86, the `in` and `out` instructions can be used to communicate with devices. For example, to send data to a device, the caller specifies a register with the data in it, and a specific *port* which names the device. Executing the instruction leads to the desired behavior.

Such instructions are usually **privileged**. The OS controls devices, and the OS thus is the only entity allowed to directly communicate with them. Imagine if any program could read or write the disk, for example: total chaos (as always), as any user program could use such a loophole to gain complete control over the machine.

The second method to interact with devices is known as **memory-mapped I/O**. With this approach, the hardware makes device registers available as if they were memory locations. To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

There is not some great advantage to one approach or the other. The memory-mapped approach is nice in that no new instructions are needed to support it, but both approaches are still in use today.

36.7 Fitting Into The OS: The Device Driver

One final problem we will discuss: how to fit devices, each of which have very specific interfaces, into the OS, which we would like to keep as general as possible. For example, consider a file system. We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drives, and so forth, and we'd like the file system to be relatively oblivious to all of the details of how to issue a read or write request to these different types of drives. Thus, our problem:

THE CRUX: HOW TO BUILD A DEVICE-NEUTRAL OS
How can we keep most of the OS device-neutral, thus hiding the details of device interactions from major OS subsystems?

The problem is solved through the age-old technique of **abstraction**. At the lowest level, a piece of software in the OS must know in detail how a device works. We call this piece of software a **device driver**, and any specifics of device interaction are encapsulated within.

Let us see how this abstraction might help OS design and implementation by examining the Linux file system software stack. Figure 36.4 is a rough and approximate depiction of the Linux software organization. As you can see from the diagram, a file system (and certainly, an application above) is completely oblivious to the specifics of which disk class it is using; it simply issues block read and write requests to the generic block layer, which routes them to the appropriate device driver, which handles the details of issuing the specific request. Although simplified, the diagram shows how such detail can be hidden from most of the OS.

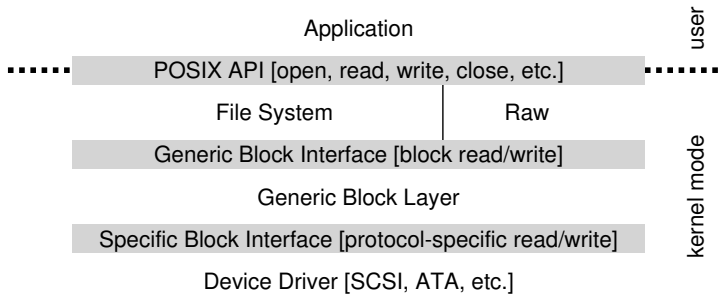


Figure 36.4: **The File System Stack**

The diagram also shows a **raw interface** to devices, which enables special applications (such as a **file-system checker**, described later [AD14], or a **disk defragmentation** tool) to directly read and write blocks without using the file abstraction. Most systems provide this type of interface to support these low-level storage management applications.

Note that the encapsulation seen above can have its downside as well. For example, if there is a device that has many special capabilities, but has to present a generic interface to the rest of the kernel, those special capabilities will go unused. This situation arises, for example, in Linux with SCSI devices, which have very rich error reporting; because other block devices (e.g., ATA/IDE) have much simpler error handling, all that higher levels of software ever receive is a generic `EIO` (generic IO error) error code; any extra detail that SCSI may have provided is thus lost to the file system [G08].

Interestingly, because device drivers are needed for any device you might plug into your system, over time they have come to represent a huge percentage of kernel code. Studies of the Linux kernel reveal that over 70% of OS code is found in device drivers [C01]; for Windows-based systems, it is likely quite high as well. Thus, when people tell you that the OS has millions of lines of code, what they are really saying is that the OS has millions of lines of device-driver code. Of course, for any given installation, most of that code may not be active (i.e., only a few devices are connected to the system at a time). Perhaps more depressingly, as drivers are often written by "amateurs" (instead of full-time kernel developers), they tend to have many more bugs and thus are a primary contributor to kernel crashes [S03].

36.8 Case Study: A Simple IDE Disk Driver

To dig a little deeper here, let's take a quick look at an actual device: an IDE disk drive [L94]. We summarize the protocol as described in this reference [W10]; we'll also peek at the xv6 source code for a simple example of a working IDE driver [CK+08].

```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset,
                  E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6     5      4     3     2     1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when ERROR==1)
    7      6     5      4     3     2     1     0
   BBK    UNC   MC   IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

Figure 36.5: **The IDE Interface**

An IDE disk presents a simple interface to the system, consisting of four types of register: control, command block, status, and error. These registers are available by reading or writing to specific "I/O addresses" (such as `0x3F6` below) using (on x86) the `in` and `out` I/O instructions.

The basic protocol to interact with the device is as follows, assuming it has already been initialized.

- **Wait for drive to be ready.** Read Status Register (0x1F7) until drive is READY and not BUSY.
- **Write parameters to command registers.** Write the sector count, logical block address (LBA) of the sectors to be accessed, and drive number (master=0x00 or slave=0x10, as IDE permits just two drives) to command registers (0x1F2-0x1F6).
- **Start the I/O.** Write READ|WRITE command to command register (0x1F7).

- **Data transfer (for writes):** Wait until drive status is READY and DRQ (drive request for data); write data to data port.
- **Handle interrupts.** In the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire transfer is complete.
- **Error handling.** After each operation, read the status register. If the ERROR bit is on, read the error register for details.

Most of this protocol is found in the xv6 IDE driver (Figure 36.6), which (after initialization) works through four primary functions. The first is `ide_rw()`, which queues a request (if there are others pending), or issues it directly to the disk (via `ide_start_request()`); in either case, the routine waits for the request to complete and the calling process is put to sleep. The second is `ide_start_request()`, which is used to send a request (and perhaps data, in the case of a write) to the disk; the `in` and `out` x86 instructions are called to read and write device registers, respectively. The start request routine uses the third function, `ide_wait_ready()`, to ensure the drive is ready before issuing a request to it. Finally, `ide_intr()` is invoked when an interrupt takes place; it reads data from the device (if the request is a read, not a write), wakes the process waiting for the I/O to complete, and (if there are more requests in the I/O queue), launches the next I/O via `ide_start_request()`.

36.9 Historical Notes

Before ending, we include a brief historical note on the origin of some of these fundamental ideas. If you are interested in learning more, read Smotherman's excellent summary [S08].

Interrupts are an ancient idea, existing on the earliest of machines. For example, the UNIVAC in the early 1950's had some form of interrupt vectoring, although it is unclear in exactly which year this feature was available [S08]. Sadly, even in its infancy, we are beginning to lose the origins of computing history.

There is also some debate as to which machine first introduced the idea of DMA. For example, Knuth and others point to the DYSEAC (a "mobile" machine, which at the time meant it could be hauled in a trailer), whereas others think the IBM SAGE may have been the first [S08]. Either way, by the mid 50's, systems with I/O devices that communicated directly with memory and interrupted the CPU when finished existed.

The history here is difficult to trace because the inventions are tied to real, and sometimes obscure, machines. For example, some think that the Lincoln Labs TX-2 machine was first with vectored interrupts [S08], but this is hardly clear.

```
static int ide_wait_ready() {
  while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
    ;  // loop until drive isn't busy
  // return -1 on error, or 0 otherwise
}

static void ide_start_request(struct buf *b) {
  ide_wait_ready();
  outb(0x3f6, 0);                 // generate interrupt
  outb(0x1f2, 1);                 // how many sectors?
  outb(0x1f3, b->sector & 0xff);  // LBA goes here ...
  outb(0x1f4, (b->sector >> 8) & 0xff);   // ... and here
  outb(0x1f5, (b->sector >> 16) & 0xff);  // ... and here!
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);    // this is a WRITE
    outsl(0x1f0, b->data, 512/4);  // transfer data too!
  } else {
    outb(0x1f7, IDE_CMD_READ);     // this is a READ (no data)
  }
}

void ide_rw(struct buf *b) {
  acquire(&ide_lock);
  for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext)
    ;                             // walk queue
  *pp = b;                        // add request to end
  if (ide_queue == b)             // if q is empty
    ide_start_request(b);         // send req to disk
  while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
    sleep(b, &ide_lock);          // wait for completion
  release(&ide_lock);
}

void ide_intr() {
  struct buf *b;
  acquire(&ide_lock);
  if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
    insl(0x1f0, b->data, 512/4);  // if READ: get data
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);                      // wake waiting process
  if ((ide_queue = b->qnext) != 0) // start next request
    ide_start_request(ide_queue);  // (if one exists)
  release(&ide_lock);
}
```

Figure 36.6: **The xv6 IDE Disk Driver (Simplified)**

Because the ideas are relatively obvious — no Einsteinian leap is required to come up with the idea of letting the CPU do something else while a slow I/O is pending — perhaps our focus on "who first?" is misguided. What is certainly clear: as people built these early machines, it became obvious that I/O support was needed. Interrupts, DMA, and related ideas are all direct outcomes of the nature of fast CPUs and slow devices; if you were there at the time, you might have had similar ideas.

36.10 Summary

You should now have a very basic understanding of how an OS interacts with a device. Two techniques, the interrupt and DMA, have been introduced to help with device efficiency, and two approaches to accessing device registers, explicit I/O instructions and memory-mapped I/O, have been described. Finally, the notion of a device driver has been presented, showing how the OS itself can encapsulate low-level details and thus make it easier to build the rest of the OS in a device-neutral fashion.

References

[A+11] "vIC: Interrupt Coalescing for Virtual Machine Storage Device IO" by Irfan Ahmad, Ajay Gulati, Ali Mashtizadeh. USENIX '11. *A terrific survey of interrupt coalescing in traditional and virtualized environments.*

[AD14] "Operating Systems: Three Easy Pieces" (Chapters: Crash Consistency: FSCK and Journaling and Log-Structured File Systems) by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. Arpaci-Dusseau Books, 2014. *A description of a file-system checker and how it works, which requires low-level access to disk devices not normally provided by the file system directly.*

[C01] "An Empirical Study of Operating System Errors" by Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, Dawson Engler. SOSP '01. *One of the first papers to systematically explore how many bugs are in modern operating systems. Among other neat findings, the authors show that device drivers have something like seven times more bugs than mainline kernel code.*

[CK+08] "The xv6 Operating System" by Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. From: http://pdos.csail.mit.edu/6.828/2008/index.html. *See `ide.c` for the IDE device driver, with a few more details therein.*

[D07] "What Every Programmer Should Know About Memory" by Ulrich Drepper. November, 2007. Available: `http://www.akkadia.org/drepper/cpumemory.pdf`. *A fantastic read about modern memory systems, starting at DRAM and going all the way up to virtualization and cache-optimized algorithms.*

[G08] "EIO: Error-handling is Occasionally Correct" by Haryadi Gunawi, Cindy Rubio-Gonzalez, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Ben Liblit. FAST '08, San Jose, CA, February 2008. *Our own work on building a tool to find code in Linux file systems that does not handle error return properly. We found hundreds and hundreds of bugs, many of which have now been fixed.*

[H17] "Intel Core i7-7700K review: Kaby Lake Debuts for Desktop" by Joel Hruska. January 3, 2017. `www.extremetech.com/extreme/241950-intels-core-i7-7700k-reviewed-kaby-lake-debuts-desktop`. *An in-depth review of a recent Intel chipset, including CPUs and the I/O subsystem.*

[H18] "Hacker News" by Many contributors. Available: https://news.ycombinator.com. *One of the better aggregators for tech-related stuff. Once back in 2014, this book became a highly-ranked entry, leading to 1 million chapter downloads in just one day! Sadly, we have yet to re-experience such a high.*

[L94] "AT Attachment Interface for Disk Drives" by Lawrence J. Lamers. Reference number: ANSI X3.221, 1994. Available: `ftp://ftp.t10.org/t13/project/d0791r4c-ATA-1.pdf`. *A rather dry document about device interfaces. Read it at your own peril.*

[MR96] "Eliminating Receive Livelock in an Interrupt-driven Kernel" by Jeffrey Mogul, K. K. Ramakrishnan. USENIX '96, San Diego, CA, January 1996. *Mogul and colleagues did a great deal of pioneering work on web server network performance. This paper is but one example.*

[S08] "Interrupts" by Mark Smotherman. July '08. Available: `http://people.cs.clemson.edu/~mark/interrupts.html`. *A treasure trove of information on the history of interrupts, DMA, and related early ideas in computing.*

[S03] "Improving the Reliability of Commodity Operating Systems" by Michael M. Swift, Brian N. Bershad, Henry M. Levy. SOSP '03. *Swift's work revived interest in a more microkernel-like approach to operating systems; minimally, it finally gave some good reasons why address-space based protection could be useful in a modern OS.*

[W10] "Hard Disk Driver" by Washington State Course Homepage. Available online at this site: `http://eecs.wsu.edu/~cs460/cs560/HDdriver.html`. *A nice summary of a simple IDE disk drive's interface and how to build a device driver for it.*

37

Hard Disk Drives

The last chapter introduced the general concept of an I/O device and showed you how the OS might interact with such a beast. In this chapter, we dive into more detail about one device in particular: the **hard disk drive**. These drives have been the main form of persistent data storage in computer systems for decades and much of the development of file system technology (coming soon) is predicated on their behavior. Thus, it is worth understanding the details of a disk's operation before building the file system software that manages it. Many of these details are available in excellent papers by Ruemmler and Wilkes [RW92] and Anderson, Dykes, and Riedel [ADR03].

CRUX: HOW TO STORE AND ACCESS DATA ON DISK
How do modern hard-disk drives store data? What is the interface? How is the data actually laid out and accessed? How does disk scheduling improve performance?

37.1 The Interface

Let's start by understanding the interface to a modern disk drive. The basic interface for all modern drives is straightforward. The drive consists of a large number of sectors (512-byte blocks), each of which can be read or written. The sectors are numbered from 0 to $n - 1$ on a disk with n sectors. Thus, we can view the disk as an array of sectors; 0 to $n - 1$ is thus the **address space** of the drive.

Multi-sector operations are possible; indeed, many file systems will read or write 4KB at a time (or more). However, when updating the disk, the only guarantee drive manufacturers make is that a single 512-byte write is **atomic** (i.e., it will either complete in its entirety or it won't complete at all); thus, if an untimely power loss occurs, only a portion of a larger write may complete (sometimes called a **torn write**).

There are some assumptions most clients of disk drives make, but that are not specified directly in the interface; Schlosser and Ganger have

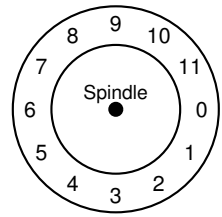


Figure 37.1: **A Disk With Just A Single Track**

called this the "unwritten contract" of disk drives [SG04]. Specifically, one can usually assume that accessing two blocks[1] near one-another within the drive's address space will be faster than accessing two blocks that are far apart. One can also usually assume that accessing blocks in a contiguous chunk (i.e., a sequential read or write) is the fastest access mode, and usually much faster than any more random access pattern.

37.2 Basic Geometry

Let's start to understand some of the components of a modern disk. We start with a **platter**, a circular hard surface on which data is stored persistently by inducing magnetic changes to it. A disk may have one or more platters; each platter has 2 sides, each of which is called a **surface**. These platters are usually made of some hard material (such as aluminum), and then coated with a thin magnetic layer that enables the drive to persistently store bits even when the drive is powered off.

The platters are all bound together around the **spindle**, which is connected to a motor that spins the platters around (while the drive is powered on) at a constant (fixed) rate. The rate of rotation is often measured in **rotations per minute (RPM)**, and typical modern values are in the 7,200 RPM to 15,000 RPM range. Note that we will often be interested in the time of a single rotation, e.g., a drive that rotates at 10,000 RPM means that a single rotation takes about 6 milliseconds (6 ms).

Data is encoded on each surface in concentric circles of sectors; we call one such concentric circle a **track**. A single surface contains many thousands and thousands of tracks, tightly packed together, with hundreds of tracks fitting into the width of a human hair.

To read and write from the surface, we need a mechanism that allows us to either sense (i.e., read) the magnetic patterns on the disk or to induce a change in (i.e., write) them. This process of reading and writing is accomplished by the **disk head**; there is one such head per surface of the drive. The disk head is attached to a single **disk arm**, which moves across the surface to position the head over the desired track.

[1] We, and others, often use the terms **block** and **sector** interchangeably, assuming the reader will know exactly what is meant per context. Sorry about this!

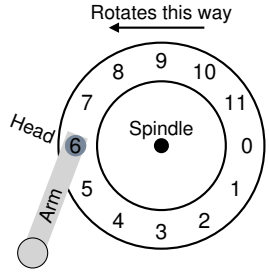


Figure 37.2: **A Single Track Plus A Head**

37.3 A Simple Disk Drive

Let's understand how disks work by building up a model one track at a time. Assume we have a simple disk with a single track (Figure 37.1). This track has just 12 sectors, each of which is 512 bytes in size (our typical sector size, recall) and addressed therefore by the numbers 0 through 11. The single platter we have here rotates around the spindle, to which a motor is attached.

Of course, the track by itself isn't too interesting; we want to be able to read or write those sectors, and thus we need a disk head, attached to a disk arm, as we now see (Figure 37.2). In the figure, the disk head, attached to the end of the arm, is positioned over sector 6, and the surface is rotating counter-clockwise.

Single-track Latency: The Rotational Delay

To understand how a request would be processed on our simple, one-track disk, imagine we now receive a request to read block 0. How should the disk service this request?

In our simple disk, the disk doesn't have to do much. In particular, it must just wait for the desired sector to rotate under the disk head. This wait happens often enough in modern drives, and is an important enough component of I/O service time, that it has a special name: **rotational delay** (sometimes **rotation delay**, though that sounds weird). In the example, if the full rotational delay is R, the disk has to incur a rotational delay of about $\frac{R}{2}$ to wait for 0 to come under the read/write head (if we start at 6). A worst-case request on this single track would be to sector 5, causing nearly a full rotational delay in order to service such a request.

Multiple Tracks: Seek Time

So far our disk just has a single track, which is not too realistic; modern disks of course have many millions. Let's thus look at an ever-so-slightly more realistic disk surface, this one with three tracks (Figure 37.3, left).

In the figure, the head is currently positioned over the innermost track (which contains sectors 24 through 35); the next track over contains the

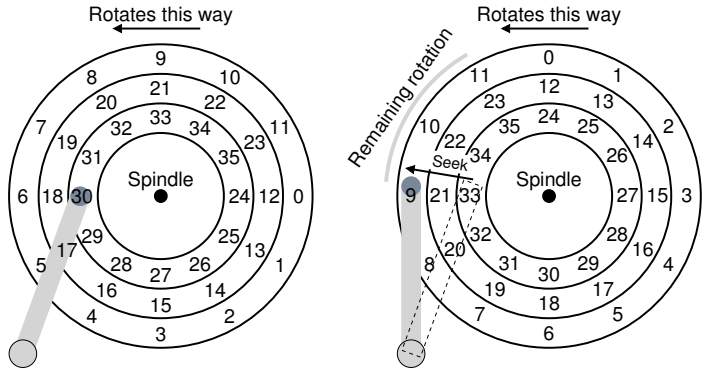


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

next set of sectors (12 through 23), and the outermost track contains the first sectors (0 through 11).

To understand how the drive might access a given sector, we now trace what would happen on a request to a distant sector, e.g., a read to sector 11. To service this read, the drive has to first move the disk arm to the correct track (in this case, the outermost one), in a process known as a **seek**. Seeks, along with rotations, are one of the most costly disk operations.

The seek, it should be noted, has many phases: first an *acceleration* phase as the disk arm gets moving; then *coasting* as the arm is moving at full speed, then *deceleration* as the arm slows down; finally *settling* as the head is carefully positioned over the correct track. The **settling time** is often quite significant, e.g., 0.5 to 2 ms, as the drive must be certain to find the right track (imagine if it just got close instead!).

After the seek, the disk arm has positioned the head over the right track. A depiction of the seek is found in Figure 37.3 (right).

As we can see, during the seek, the arm has been moved to the desired track, and the platter of course has rotated, in this case about 3 sectors. Thus, sector 9 is just about to pass under the disk head, and we must only endure a short rotational delay to complete the transfer.

When sector 11 passes under the disk head, the final phase of I/O will take place, known as the **transfer**, where data is either read from or written to the surface. And thus, we have a complete picture of I/O time: first a seek, then waiting for the rotational delay, and finally the transfer.

Some Other Details

Though we won't spend too much time on it, there are some other interesting details about how hard drives operate. Many drives employ some kind of **track skew** to make sure that sequential reads can be properly serviced even when crossing track boundaries. In our simple example disk, this might appear as seen in Figure 37.4 (page 5).

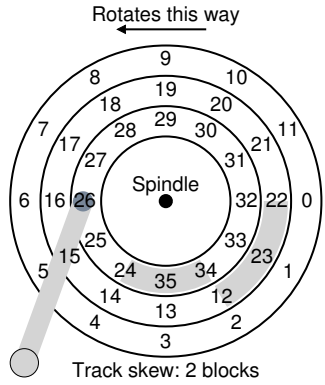


Figure 37.4: **Three Tracks: Track Skew Of 2**

Sectors are often skewed like this because when switching from one track to another, the disk needs time to reposition the head (even to neighboring tracks). Without such skew, the head would be moved to the next track but the desired next block would have already rotated under the head, and thus the drive would have to wait almost the entire rotational delay to access the next block.

Another reality is that outer tracks tend to have more sectors than inner tracks, which is a result of geometry; there is simply more room out there. These tracks are often referred to as **multi-zoned** disk drives, where the disk is organized into multiple zones, and where a zone is consecutive set of tracks on a surface. Each zone has the same number of sectors per track, and outer zones have more sectors than inner zones.

Finally, an important part of any modern disk drive is its **cache**, for historical reasons sometimes called a **track buffer**. This cache is just some small amount of memory (usually around 8 or 16 MB) which the drive can use to hold data read from or written to the disk. For example, when reading a sector from the disk, the drive might decide to read in all of the sectors on that track and cache them in its memory; doing so allows the drive to quickly respond to any subsequent requests to the same track.

On writes, the drive has a choice: should it acknowledge the write has completed when it has put the data in its memory, or after the write has actually been written to disk? The former is called **write back** caching (or sometimes **immediate reporting**), and the latter **write through**. Write back caching sometimes makes the drive appear "faster", but can be dangerous; if the file system or applications require that data be written to disk in a certain order for correctness, write-back caching can lead to problems (read the chapter on file-system journaling for details).

ASIDE: DIMENSIONAL ANALYSIS

Remember in Chemistry class, how you solved virtually every problem by simply setting up the units such that they canceled out, and somehow the answers popped out as a result? That chemical magic is known by the highfalutin name of **dimensional analysis** and it turns out it is useful in computer systems analysis too.

Let's do an example to see how dimensional analysis works and why it is useful. In this case, assume you have to figure out how long, in milliseconds, a single rotation of a disk takes. Unfortunately, you are given only the **RPM** of the disk, or **rotations per minute**. Let's assume we're talking about a 10K RPM disk (i.e., it rotates 10,000 times per minute). How do we set up the dimensional analysis so that we get time per rotation in milliseconds?

To do so, we start by putting the desired units on the left; in this case, we wish to obtain the time (in milliseconds) per rotation, so that is exactly what we write down: $\frac{Time\ (ms)}{1\ Rotation}$. We then write down everything we know, making sure to cancel units where possible. First, we obtain $\frac{1\ minute}{10{,}000\ Rotations}$ (keeping rotation on the bottom, as that's where it is on the left), then transform minutes into seconds with $\frac{60\ seconds}{1\ minute}$, and then finally transform seconds in milliseconds with $\frac{1000\ ms}{1\ second}$. The final result is the following (with units nicely canceled):

$$\frac{Time\ (ms)}{1\ Rot.} = \frac{1\ \cancel{minute}}{10{,}000\ Rot.} \cdot \frac{60\ \cancel{seconds}}{1\ \cancel{minute}} \cdot \frac{1000\ ms}{1\ \cancel{second}} = \frac{60{,}000\ ms}{10{,}000\ Rot.} = \frac{6\ ms}{Rotation}$$

As you can see from this example, dimensional analysis makes what seems intuitive into a simple and repeatable process. Beyond the RPM calculation above, it comes in handy with I/O analysis regularly. For example, you will often be given the transfer rate of a disk, e.g., 100 MB/second, and then asked: how long does it take to transfer a 512 KB block (in milliseconds)? With dimensional analysis, it's easy:

$$\frac{Time\ (ms)}{1\ Request} = \frac{512\ \cancel{KB}}{1\ Request} \cdot \frac{1\ \cancel{MB}}{1024\ \cancel{KB}} \cdot \frac{1\ \cancel{second}}{100\ \cancel{MB}} \cdot \frac{1000\ ms}{1\ \cancel{second}} = \frac{5\ ms}{Request}$$

37.4 I/O Time: Doing The Math

Now that we have an abstract model of the disk, we can use a little analysis to better understand disk performance. In particular, we can now represent I/O time as the sum of three major components:

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer} \tag{37.1}$$

Note that the rate of I/O ($R_{I/O}$), which is often more easily used for

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

Figure 37.5: **Disk Drive Specs: SCSI Versus SATA**

comparison between drives (as we will do below), is easily computed from the time. Simply divide the size of the transfer by the time it took:

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}} \tag{37.2}$$

To get a better feel for I/O time, let us perform the following calculation. Assume there are two workloads we are interested in. The first, known as the **random** workload, issues small (e.g., 4KB) reads to random locations on the disk. Random workloads are common in many important applications, including database management systems. The second, known as the **sequential** workload, simply reads a large number of sectors consecutively from the disk, without jumping around. Sequential access patterns are quite common and thus important as well.

To understand the difference in performance between random and sequential workloads, we need to make a few assumptions about the disk drive first. Let's look at a couple of modern disks from Seagate. The first, known as the Cheetah 15K.5 [S09b], is a high-performance SCSI drive. The second, the Barracuda [S09a], is a drive built for capacity. Details on both are found in Figure 37.5.

As you can see, the drives have quite different characteristics, and in many ways nicely summarize two important components of the disk drive market. The first is the "high performance" drive market, where drives are engineered to spin as fast as possible, deliver low seek times, and transfer data quickly. The second is the "capacity" market, where cost per byte is the most important aspect; thus, the drives are slower but pack as many bits as possible into the space available.

From these numbers, we can start to calculate how well the drives would do under our two workloads outlined above. Let's start by looking at the random workload. Assuming each 4 KB read occurs at a random location on disk, we can calculate how long each such read would take. On the Cheetah:

$$T_{seek} = 4\ ms,\ T_{rotation} = 2\ ms,\ T_{transfer} = 30\ microsecs \tag{37.3}$$

The average seek time (4 milliseconds) is just taken as the average time reported by the manufacturer; note that a full seek (from one end of the

TIP: USE DISKS SEQUENTIALLY
When at all possible, transfer data to and from disks in a sequential manner. If sequential is not possible, at least think about transferring data in large chunks: the bigger, the better. If I/O is done in little random pieces, I/O performance will suffer dramatically. Also, users will suffer. Also, you will suffer, knowing what suffering you have wrought with your careless random I/Os.

surface to the other) would likely take two or three times longer. The average rotational delay is calculated from the RPM directly. 15000 RPM is equal to 250 RPS (rotations per second); thus, each rotation takes 4 ms. On average, the disk will encounter a half rotation and thus 2 ms is the average time. Finally, the transfer time is just the size of the transfer over the peak transfer rate; here it is vanishingly small (30 *microseconds*; note that we need 1000 microseconds just to get 1 millisecond!).

Thus, from our equation above, $T_{I/O}$ for the Cheetah roughly equals 6 ms. To compute the rate of I/O, we just divide the size of the transfer by the average time, and thus arrive at $R_{I/O}$ for the Cheetah under the random workload of about 0.66 MB/s. The same calculation for the Barracuda yields a $T_{I/O}$ of about 13.2 ms, more than twice as slow, and thus a rate of about 0.31 MB/s.

Now let's look at the sequential workload. Here we can assume there is a single seek and rotation before a very long transfer. For simplicity, assume the size of the transfer is 100 MB. Thus, $T_{I/O}$ for the Cheetah and Barracuda is about 800 ms and 950 ms, respectively. The rates of I/O are thus very nearly the peak transfer rates of 125 MB/s and 105 MB/s, respectively. Figure 37.6 summarizes these numbers.

	Cheetah	Barracuda
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

Figure 37.6: **Disk Drive Performance: SCSI Versus SATA**

The figure shows us a number of important things. First, and most importantly, there is a huge gap in drive performance between random and sequential workloads, almost a factor of 200 or so for the Cheetah and more than a factor 300 difference for the Barracuda. And thus we arrive at the most obvious design tip in the history of computing.

A second, more subtle point: there is a large difference in performance between high-end "performance" drives and low-end "capacity" drives. For this reason (and others), people are often willing to pay top dollar for the former while trying to get the latter as cheaply as possible.

ASIDE: COMPUTING THE "AVERAGE" SEEK

In many books and papers, you will see average disk-seek time cited as being roughly one-third of the full seek time. Where does this come from?

Turns out it arises from a simple calculation based on average seek *distance*, not time. Imagine the disk as a set of tracks, from 0 to N. The seek distance between any two tracks x and y is thus computed as the absolute value of the difference between them: $|x - y|$.

To compute the average seek distance, all you need to do is to first add up all possible seek distances:

$$\sum_{x=0}^{N}\sum_{y=0}^{N}|x-y|. \tag{37.4}$$

Then, divide this by the number of different possible seeks: N^2. To compute the sum, we'll just use the integral form:

$$\int_{x=0}^{N}\int_{y=0}^{N}|x-y|\,\mathrm{d}y\,\mathrm{d}x. \tag{37.5}$$

To compute the inner integral, let's break out the absolute value:

$$\int_{y=0}^{x}(x-y)\,\mathrm{d}y+\int_{y=x}^{N}(y-x)\,\mathrm{d}y. \tag{37.6}$$

Solving this leads to $(xy-\frac{1}{2}y^2)\big|_0^x+(\frac{1}{2}y^2-xy)\big|_x^N$ which can be simplified to $(x^2-Nx+\frac{1}{2}N^2)$. Now we have to compute the outer integral:

$$\int_{x=0}^{N}(x^2-Nx+\frac{1}{2}N^2)\,\mathrm{d}x, \tag{37.7}$$

which results in:

$$(\frac{1}{3}x^3-\frac{N}{2}x^2+\frac{N^2}{2}x)\Big|_0^N=\frac{N^3}{3}. \tag{37.8}$$

Remember that we still have to divide by the total number of seeks (N^2) to compute the average seek distance: $(\frac{N^3}{3})/(N^2)=\frac{1}{3}N$. Thus the average seek distance on a disk, over all possible seeks, is one-third the full distance. And now when you hear that an average seek is one-third of a full seek, you'll know where it came from.

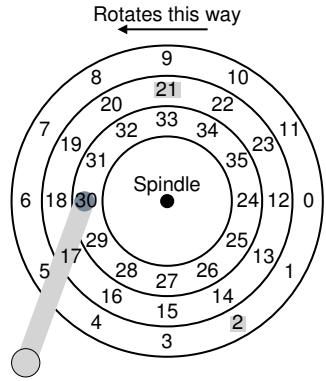


Figure 37.7: **SSTF: Scheduling Requests 21 And 2**

37.5 Disk Scheduling

Because of the high cost of I/O, the OS has historically played a role in deciding the order of I/Os issued to the disk. More specifically, given a set of I/O requests, the **disk scheduler** examines the requests and decides which one to schedule next [SCO90, JW91].

Unlike job scheduling, where the length of each job is usually unknown, with disk scheduling, we can make a good guess at how long a "job" (i.e., disk request) will take. By estimating the seek and possible rotational delay of a request, the disk scheduler can know how long each request will take, and thus (greedily) pick the one that will take the least time to service first. Thus, the disk scheduler will try to follow the **principle of SJF (shortest job first)** in its operation.

SSTF: Shortest Seek Time First

One early disk scheduling approach is known as **shortest-seek-time-first** (**SSTF**) (also called **shortest-seek-first** or **SSF**). SSTF orders the queue of I/O requests by track, picking requests on the nearest track to complete first. For example, assuming the current position of the head is over the inner track, and we have requests for sectors 21 (middle track) and 2 (outer track), we would then issue the request to 21 first, wait for it to complete, and then issue the request to 2 (Figure 37.7).

SSTF works well in this example, seeking to the middle track first and then the outer track. However, SSTF is not a panacea, for the following reasons. First, the drive geometry is not available to the host OS; rather, it sees an array of blocks. Fortunately, this problem is rather easily fixed. Instead of SSTF, an OS can simply implement **nearest-block-first** (**NBF**), which schedules the request with the nearest block address next.

The second problem is more fundamental: **starvation**. Imagine in our example above if there were a steady stream of requests to the inner track, where the head currently is positioned. Requests to any other tracks would then be ignored completely by a pure SSTF approach. And thus the crux of the problem:

CRUX: HOW TO HANDLE DISK STARVATION
How can we implement SSTF-like scheduling but avoid starvation?

Elevator (a.k.a. SCAN or C-SCAN)

The answer to this query was developed some time ago (see [CKR72] for example), and is relatively straightforward. The algorithm, originally called **SCAN**, simply moves back and forth across the disk servicing requests in order across the tracks. Let's call a single pass across the disk (from outer to inner tracks, or inner to outer) a *sweep*. Thus, if a request comes for a block on a track that has already been serviced on this sweep of the disk, it is not handled immediately, but rather queued until the next sweep (in the other direction).

SCAN has a number of variants, all of which do about the same thing. For example, Coffman et al. introduced **F-SCAN**, which freezes the queue to be serviced when it is doing a sweep [CKR72]; this action places requests that come in during the sweep into a queue to be serviced later. Doing so avoids starvation of far-away requests, by delaying the servicing of late-arriving (but nearer by) requests.

C-SCAN is another common variant, short for **Circular SCAN**. Instead of sweeping in both directions across the disk, the algorithm only sweeps from outer-to-inner, and then resets at the outer track to begin again. Doing so is a bit more fair to inner and outer tracks, as pure back-and-forth SCAN favors the middle tracks, i.e., after servicing the outer track, SCAN passes through the middle twice before coming back to the outer track again.

For reasons that should now be clear, the SCAN algorithm (and its cousins) is sometimes referred to as the **elevator** algorithm, because it behaves like an elevator which is either going up or down and not just servicing requests to floors based on which floor is closer. Imagine how annoying it would be if you were going down from floor 10 to 1, and somebody got on at 3 and pressed 4, and the elevator went up to 4 because it was "closer" than 1! As you can see, the elevator algorithm, when used in real life, prevents fights from taking place on elevators. In disks, it just prevents starvation.

Unfortunately, SCAN and its cousins do not represent the best scheduling technology. In particular, SCAN (or SSTF even) does not actually adhere as closely to the principle of SJF as they could. In particular, they ignore rotation. And thus, another crux:

CRUX: HOW TO ACCOUNT FOR DISK ROTATION COSTS
How can we implement an algorithm that more closely approximates SJF by taking *both* seek and rotation into account?

SPTF: Shortest Positioning Time First

Before discussing **shortest positioning time first** or **SPTF** scheduling (sometimes also called **shortest access time first** or **SATF**), which is the solution to our problem, let us make sure we understand the problem in more detail. Figure 37.8 presents an example.

In the example, the head is currently positioned over sector 30 on the inner track. The scheduler thus has to decide: should it schedule sector 16 (on the middle track) or sector 8 (on the outer track) for its next request. So which should it service next?

The answer, of course, is "it depends". In engineering, it turns out "it depends" is almost always the answer, reflecting that trade-offs are part of the life of the engineer; such maxims are also good in a pinch, e.g., when you don't know an answer to your boss's question, you might want to try this gem. However, it is almost always better to know *why* it depends, which is what we discuss here.

What it depends on here is the relative time of seeking as compared to rotation. If, in our example, seek time is much higher than rotational delay, then SSTF (and variants) are just fine. However, imagine if seek is quite a bit faster than rotation. Then, in our example, it would make more sense to seek *further* to service request 8 on the outer track than it would to perform the shorter seek to the middle track to service 16, which has to rotate all the way around before passing under the disk head.

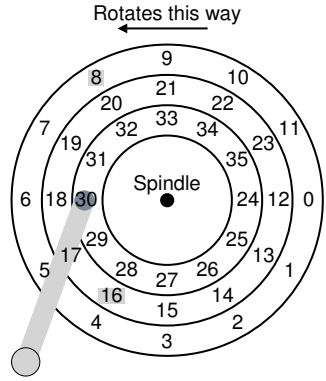


Figure 37.8: **SSTF: Sometimes Not Good Enough**

TIP: IT ALWAYS DEPENDS (LIVNY'S LAW)
Almost any question can be answered with "it depends", as our colleague Miron Livny always says. However, use with caution, as if you answer too many questions this way, people will stop asking you questions altogether. For example, somebody asks: "want to go to lunch?" You reply: "it depends, are *you* coming along?"

On modern drives, as we saw above, both seek and rotation are roughly equivalent (depending, of course, on the exact requests), and thus SPTF is useful and improves performance. However, it is even more difficult to implement in an OS, which generally does not have a good idea where track boundaries are or where the disk head currently is (in a rotational sense). Thus, SPTF is usually performed inside a drive, described below.

Other Scheduling Issues

There are many other issues we do not discuss in this brief description of basic disk operation, scheduling, and related topics. One such issue is this: *where* is disk scheduling performed on modern systems? In older systems, the operating system did all the scheduling; after looking through the set of pending requests, the OS would pick the best one, and issue it to the disk. When that request completed, the next one would be chosen, and so forth. Disks were simpler then, and so was life.

In modern systems, disks can accommodate multiple outstanding requests, and have sophisticated internal schedulers themselves (which can implement SPTF accurately; inside the disk controller, all relevant details are available, including exact head position). Thus, the OS scheduler usually picks what it thinks the best few requests are (say 16) and issues them all to disk; the disk then uses its internal knowledge of head position and detailed track layout information to service said requests in the best possible (SPTF) order.

Another important related task performed by disk schedulers is **I/O merging**. For example, imagine a series of requests to read blocks 33, then 8, then 34, as in Figure 37.8. In this case, the scheduler should **merge** the requests for blocks 33 and 34 into a single two-block request; any reordering that the scheduler does is performed upon the merged requests. Merging is particularly important at the OS level, as it reduces the number of requests sent to the disk and thus lowers overheads.

One final problem that modern schedulers address is this: how long should the system wait before issuing an I/O to disk? One might naively think that the disk, once it has even a single I/O, should immediately issue the request to the drive; this approach is called **work-conserving**, as the disk will never be idle if there are requests to serve. However, research on **anticipatory disk scheduling** has shown that sometimes it is better to

wait for a bit [ID01], in what is called a **non-work-conserving** approach. By waiting, a new and "better" request may arrive at the disk, and thus overall efficiency is increased. Of course, deciding when to wait, and for how long, can be tricky; see the research paper for details, or check out the Linux kernel implementation to see how such ideas are transitioned into practice (if you are the ambitious sort).

37.6 Summary

We have presented a summary of how disks work. The summary is actually a detailed functional model; it does not describe the amazing physics, electronics, and material science that goes into actual drive design. For those interested in even more details of that nature, we suggest a different major (or perhaps minor); for those that are happy with this model, good! We can now proceed to using the model to build more interesting systems on top of these incredible devices.

References

[ADR03] "More Than an Interface: SCSI vs. ATA" by Dave Anderson, Jim Dykes, Erik Riedel. FAST '03, 2003. *One of the best recent-ish references on how modern disk drives really work; a must read for anyone interested in knowing more.*

[CKR72] "Analysis of Scanning Policies for Reducing Disk Seek Times" E.G. Coffman, L.A. Klimko, B. Ryan SIAM Journal of Computing, September 1972, Vol 1. No 3. *Some of the early work in the field of disk scheduling.*

[HK+17] "The Unwritten Contract of Solid State Drives" by Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. EuroSys '17, Belgrade, Serbia, April 2017. *We take the idea of the unwritten contract, and extend it to SSDs. Using SSDs well seems as complicated as hard drives, and sometimes more so.*

[ID01] "Anticipatory Scheduling: A Disk-scheduling Framework To Overcome Deceptive Idleness In Synchronous I/O" by Sitaram Iyer, Peter Druschel. SOSP '01, October 2001. *A cool paper showing how waiting can improve disk scheduling: better requests may be on their way!*

[JW91] "Disk Scheduling Algorithms Based On Rotational Position" by D. Jacobson, J. Wilkes. Technical Report HPL-CSP-91-7rev1, Hewlett-Packard, February 1991. *A more modern take on disk scheduling. It remains a technical report (and not a published paper) because the authors were scooped by Seltzer et al. [SCO90].*

[RW92] "An Introduction to Disk Drive Modeling" by C. Ruemmler, J. Wilkes. IEEE Computer, 27:3, March 1994. *A terrific introduction to the basics of disk operation. Some pieces are out of date, but most of the basics remain.*

[SCO90] "Disk Scheduling Revisited" by Margo Seltzer, Peter Chen, John Ousterhout. USENIX 1990. *A paper that talks about how rotation matters too in the world of disk scheduling.*

[SG04] "MEMS-based storage devices and standard disk interfaces: A square peg in a round hole?" Steven W. Schlosser, Gregory R. Ganger FAST '04, pp. 87-100, 2004 *While the MEMS aspect of this paper hasn't yet made an impact, the discussion of the contract between file systems and disks is wonderful and a lasting contribution. We later build on this work to study the "Unwritten Contract of Solid State Drives" [HK+17]*

[S09a] "Barracuda ES.2 data sheet" by Seagate, Inc.. Available at this website, at least, it was: `http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es.pdf`. *A data sheet; read at your own risk. Risk of what? Boredom.*

[S09b] "Cheetah 15K.5" by Seagate, Inc.. Available at this website, we're pretty sure it is: `http://www.seagate.com/docs/pdf/datasheet/disc/ds-cheetah-15k-5-us.pdf`. *See above commentary on data sheets.*

Homework (Simulation)

This homework uses `disk.py` to familiarize you with how a modern hard drive works. It has a lot of different options, and unlike most of the other simulations, has a graphical animator to show you exactly what happens when the disk is in action. See the README for details.

1. Compute the seek, rotation, and transfer times for the following sets of requests: `-a 0`, `-a 6`, `-a 30`, `-a 7,30,8`, and finally `-a 10,11,12,13`.
2. Do the same requests above, but change the seek rate to different values: `-S 2`, `-S 4`, `-S 8`, `-S 10`, `-S 40`, `-S 0.1`. How do the times change?
3. Do the same requests above, but change the rotation rate: `-R 0.1`, `-R 0.5`, `-R 0.01`. How do the times change?
4. FIFO is not always best, e.g., with the request stream `-a 7,30,8`, what order should the requests be processed in? Run the shortest seek-time first (SSTF) scheduler (`-p SSTF`) on this workload; how long should it take (seek, rotation, transfer) for each request to be served?
5. Now use the shortest access-time first (SATF) scheduler (`-p SATF`). Does it make any difference for `-a 7,30,8` workload? Find a set of requests where SATF outperforms SSTF; more generally, when is SATF better than SSTF?
6. Here is a request stream to try: `-a 10,11,12,13`. What goes poorly when it runs? Try adding track skew to address this problem (`-o skew`). Given the default seek rate, what should the skew be to maximize performance? What about for different seek rates (e.g., `-S 2`, `-S 4`)? In general, could you write a formula to figure out the skew?
7. Specify a disk with different density per zone, e.g., `-z 10,20,30`, which specifies the angular difference between blocks on the outer, middle, and inner tracks. Run some random requests (e.g., `-a -1 -A 5,-1,0`, which specifies that random requests should be used via the `-a -1` flag and that five requests ranging from 0 to the max be generated), and compute the seek, rotation, and transfer times. Use different random seeds. What is the bandwidth (in sectors per unit time) on the outer, middle, and inner tracks?
8. A scheduling window determines how many requests the disk can examine at once. Generate random workloads (e.g., `-A 1000,-1,0`, with different seeds) and see how long the SATF scheduler takes when the scheduling window is changed from 1 up to the number of requests. How big of a window is needed to maximize performance? Hint: use the `-c` flag and don't turn on graphics (`-G`) to run these quickly. When the scheduling window is set to 1, does it matter which policy you are using?
9. Create a series of requests to starve a particular request, assuming an SATF policy. Given that sequence, how does it perform if you use a **bounded SATF** (**BSATF**) scheduling approach? In this approach, you specify the scheduling window (e.g., `-w 4`); the scheduler only moves onto the next window of requests when *all* requests in the current window have been serviced. Does this solve starvation? How does it perform, as compared to SATF? In general, how should a disk make this trade-off between performance and starvation avoidance?
10. All the scheduling policies we have looked at thus far are **greedy**; they pick the next best option instead of looking for an optimal schedule. Can you find a set of requests in which greedy is not optimal?

38

Redundant Arrays of Inexpensive Disks (RAIDs)

When we use a disk, we sometimes wish it to be faster; I/O operations are slow and thus can be the bottleneck for the entire system. When we use a disk, we sometimes wish it to be larger; more and more data is being put online and thus our disks are getting fuller and fuller. When we use a disk, we sometimes wish for it to be more reliable; when a disk fails, if our data isn't backed up, all that valuable data is gone.

CRUX: HOW TO MAKE A LARGE, FAST, RELIABLE DISK
How can we make a large, fast, and reliable storage system? What are the key techniques? What are trade-offs between different approaches?

In this chapter, we introduce the **Redundant Array of Inexpensive Disks** better known as **RAID** [P+88], a technique to use multiple disks in concert to build a faster, bigger, and more reliable disk system. The term was introduced in the late 1980s by a group of researchers at U.C. Berkeley (led by Professors David Patterson and Randy Katz and then student Garth Gibson); it was around this time that many different researchers simultaneously arrived upon the basic idea of using multiple disks to build a better storage system [BG88, K86,K88,PB86,SG86].

Externally, a RAID looks like a disk: a group of blocks one can read or write. Internally, the RAID is a complex beast, consisting of multiple disks, memory (both volatile and non-), and one or more processors to manage the system. A hardware RAID is very much like a computer system, specialized for the task of managing a group of disks.

RAIDs offer a number of advantages over a single disk. One advantage is *performance*. Using multiple disks in parallel can greatly speed up I/O times. Another benefit is *capacity*. Large data sets demand large disks. Finally, RAIDs can improve *reliability*; spreading data across multiple disks (without RAID techniques) makes the data vulnerable to the loss of a single disk; with some form of **redundancy**, RAIDs can tolerate the loss of a disk and keep operating as if nothing were wrong.

TIP: TRANSPARENCY ENABLES DEPLOYMENT
When considering how to add new functionality to a system, one should always consider whether such functionality can be added **transparently**, in a way that demands no changes to the rest of the system. Requiring a complete rewrite of the existing software (or radical hardware changes) lessens the chance of impact of an idea. RAID is a perfect example, and certainly its transparency contributed to its success; administrators could install a SCSI-based RAID storage array instead of a SCSI disk, and the rest of the system (host computer, OS, etc.) did not have to change one bit to start using it. By solving this problem of **deployment**, RAID was made more successful from day one.

Amazingly, RAIDs provide these advantages **transparently** to systems that use them, i.e., a RAID just looks like a big disk to the host system. The beauty of transparency, of course, is that it enables one to simply replace a disk with a RAID and not change a single line of software; the operating system and client applications continue to operate without modification. In this manner, transparency greatly improves the **deployability** of RAID, enabling users and administrators to put a RAID to use without worries of software compatibility.

We now discuss some of the important aspects of RAIDs. We begin with the interface, fault model, and then discuss how one can evaluate a RAID design along three important axes: capacity, reliability, and performance. We then discuss a number of other issues that are important to RAID design and implementation.

38.1 Interface And RAID Internals

To a file system above, a RAID looks like a big, (hopefully) fast, and (hopefully) reliable disk. Just as with a single disk, it presents itself as a linear array of blocks, each of which can be read or written by the file system (or other client).

When a file system issues a *logical I/O* request to the RAID, the RAID internally must calculate which disk (or disks) to access in order to complete the request, and then issue one or more *physical I/Os* to do so. The exact nature of these physical I/Os depends on the RAID level, as we will discuss in detail below. However, as a simple example, consider a RAID that keeps two copies of each block (each one on a separate disk); when writing to such a **mirrored** RAID system, the RAID will have to perform two physical I/Os for every one logical I/O it is issued.

A RAID system is often built as a separate hardware box, with a standard connection (e.g., SCSI, or SATA) to a host. Internally, however, RAIDs are fairly complex, consisting of a microcontroller that runs firmware to direct the operation of the RAID, volatile memory such as DRAM to buffer data blocks as they are read and written, and in some cases,

non-volatile memory to buffer writes safely and perhaps even specialized logic to perform parity calculations (useful in some RAID levels, as we will also see below). At a high level, a RAID is very much a specialized computer system: it has a processor, memory, and disks; however, instead of running applications, it runs specialized software designed to operate the RAID.

38.2 Fault Model

To understand RAID and compare different approaches, we must have a fault model in mind. RAIDs are designed to detect and recover from certain kinds of disk faults; thus, knowing exactly which faults to expect is critical in arriving upon a working design.

The first fault model we will assume is quite simple, and has been called the **fail-stop** fault model [S84]. In this model, a disk can be in exactly one of two states: working or failed. With a working disk, all blocks can be read or written. In contrast, when a disk has failed, we assume it is permanently lost.

One critical aspect of the fail-stop model is what it assumes about fault detection. Specifically, when a disk has failed, we assume that this is easily detected. For example, in a RAID array, we would assume that the RAID controller hardware (or software) can immediately observe when a disk has failed.

Thus, for now, we do not have to worry about more complex "silent" failures such as disk corruption. We also do not have to worry about a single block becoming inaccessible upon an otherwise working disk (sometimes called a latent sector error). We will consider these more complex (and unfortunately, more realistic) disk faults later.

38.3 How To Evaluate A RAID

As we will soon see, there are a number of different approaches to building a RAID. Each of these approaches has different characteristics which are worth evaluating, in order to understand their strengths and weaknesses.

Specifically, we will evaluate each RAID design along three axes. The first axis is **capacity**; given a set of N disks each with B blocks, how much useful capacity is available to clients of the RAID? Without redundancy, the answer is $N \cdot B$; in contrast, if we have a system that keeps two copies of each block (called **mirroring**), we obtain a useful capacity of $(N \cdot B)/2$. Different schemes (e.g., parity-based ones) tend to fall in between.

The second axis of evaluation is **reliability**. How many disk faults can the given design tolerate? In alignment with our fault model, we assume only that an entire disk can fail; in later chapters (i.e., on data integrity), we'll think about how to handle more complex failure modes.

Finally, the third axis is **performance**. Performance is somewhat challenging to evaluate, because it depends heavily on the workload presented to the disk array. Thus, before evaluating performance, we will first present a set of typical workloads that one should consider.

We now consider three important RAID designs: RAID Level 0 (striping), RAID Level 1 (mirroring), and RAID Levels 4/5 (parity-based redundancy). The naming of each of these designs as a "level" stems from the pioneering work of Patterson, Gibson, and Katz at Berkeley [P+88].

38.4 RAID Level 0: Striping

The first RAID level is actually not a RAID level at all, in that there is no redundancy. However, RAID level 0, or **striping** as it is better known, serves as an excellent upper-bound on performance and capacity and thus is worth understanding.

The simplest form of striping will **stripe** blocks across the disks of the system as follows (assume here a 4-disk array):

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: **RAID-0: Simple Striping**

From Figure 38.1, you get the basic idea: spread the blocks of the array across the disks in a round-robin fashion. This approach is designed to extract the most parallelism from the array when requests are made for contiguous chunks of the array (as in a large, sequential read, for example). We call the blocks in the same row a **stripe**; thus, blocks 0, 1, 2, and 3 are in the same stripe above.

In the example, we have made the simplifying assumption that only 1 block (each of say size 4KB) is placed on each disk before moving on to the next. However, this arrangement need not be the case. For example, we could arrange the blocks across disks as in Figure 38.2:

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

Figure 38.2: **Striping With A Bigger Chunk Size**

In this example, we place two 4KB blocks on each disk before moving on to the next disk. Thus, the **chunk size** of this RAID array is 8KB, and a stripe thus consists of 4 chunks or 32KB of data.

ASIDE: THE RAID MAPPING PROBLEM

Before studying the capacity, reliability, and performance characteristics of the RAID, we first present an aside on what we call **the mapping problem**. This problem arises in all RAID arrays; simply put, given a logical block to read or write, how does the RAID know exactly which physical disk and offset to access?

For these simple RAID levels, we do not need much sophistication in order to correctly map logical blocks onto their physical locations. Take the first striping example above (chunk size = 1 block = 4KB). In this case, given a logical block address A, the RAID can easily compute the desired disk and offset with two simple equations:

```
Disk   = A % number_of_disks
Offset = A / number_of_disks
```

Note that these are all integer operations (e.g., 4 / 3 = 1 not 1.33333...).

Let's see how these equations work for a simple example. Imagine in the first RAID above that a request arrives for block 14. Given that there are 4 disks, this would mean that the disk we are interested in is (14 % 4 = 2): disk 2. The exact block is calculated as (14 / 4 = 3): block 3. Thus, block 14 should be found on the fourth block (block 3, starting at 0) of the third disk (disk 2, starting at 0), which is exactly where it is.

You can think about how these equations would be modified to support different chunk sizes. Try it! It's not too hard.

Chunk Sizes

Chunk size mostly affects performance of the array. For example, a small chunk size implies that many files will get striped across many disks, thus increasing the parallelism of reads and writes to a single file; however, the positioning time to access blocks across multiple disks increases, because the positioning time for the entire request is determined by the maximum of the positioning times of the requests across all drives.

A big chunk size, on the other hand, reduces such intra-file parallelism, and thus relies on multiple concurrent requests to achieve high throughput. However, large chunk sizes reduce positioning time; if, for example, a single file fits within a chunk and thus is placed on a single disk, the positioning time incurred while accessing it will just be the positioning time of a single disk.

Thus, determining the "best" chunk size is hard to do, as it requires a great deal of knowledge about the workload presented to the disk system [CL95]. For the rest of this discussion, we will assume that the array uses a chunk size of a single block (4KB). Most arrays use larger chunk sizes (e.g., 64 KB), but for the issues we discuss below, the exact chunk size does not matter; thus we use a single block for the sake of simplicity.

Back To RAID-0 Analysis

Let us now evaluate the capacity, reliability, and performance of striping. From the perspective of capacity, it is perfect: given N disks each of size B blocks, striping delivers $N \cdot B$ blocks of useful capacity. From the standpoint of reliability, striping is also perfect, but in the bad way: any disk failure will lead to data loss. Finally, performance is excellent: all disks are utilized, often in parallel, to service user I/O requests.

Evaluating RAID Performance

In analyzing RAID performance, one can consider two different performance metrics. The first is *single-request latency*. Understanding the latency of a single I/O request to a RAID is useful as it reveals how much parallelism can exist during a single logical I/O operation. The second is *steady-state throughput* of the RAID, i.e., the total bandwidth of many concurrent requests. Because RAIDs are often used in high-performance environments, the steady-state bandwidth is critical, and thus will be the main focus of our analyses.

To understand throughput in more detail, we need to put forth some workloads of interest. We will assume, for this discussion, that there are two types of workloads: **sequential** and **random**. With a sequential workload, we assume that requests to the array come in large contiguous chunks; for example, a request (or series of requests) that accesses 1 MB of data, starting at block x and ending at block (x+1 MB), would be deemed sequential. Sequential workloads are common in many environments (think of searching through a large file for a keyword), and thus are considered important.

For random workloads, we assume that each request is rather small, and that each request is to a different random location on disk. For example, a random stream of requests may first access 4KB at logical address 10, then at logical address 550,000, then at 20,100, and so forth. Some important workloads, such as transactional workloads on a database management system (DBMS), exhibit this type of access pattern, and thus it is considered an important workload.

Of course, real workloads are not so simple, and often have a mix of sequential and random-seeming components as well as behaviors in-between the two. For simplicity, we just consider these two possibilities.

As you can tell, sequential and random workloads will result in widely different performance characteristics from a disk. With sequential access, a disk operates in its most efficient mode, spending little time seeking and waiting for rotation and most of its time transferring data. With random access, just the opposite is true: most time is spent seeking and waiting for rotation and relatively little time is spent transferring data. To capture this difference in our analysis, we will assume that a disk can transfer data at S MB/s under a sequential workload, and R MB/s when under a random workload. In general, S is much greater than R (i.e., $S \gg R$).

To make sure we understand this difference, let's do a simple exercise. Specifically, let's calculate S and R given the following disk characteristics. Assume a sequential transfer of size 10 MB on average, and a random transfer of 10 KB on average. Also, assume the following disk characteristics:

Average seek time	7 ms
Average rotational delay	3 ms
Transfer rate of disk	50 MB/s

To compute S, we need to first figure out how time is spent in a typical 10 MB transfer. First, we spend 7 ms seeking, and then 3 ms rotating. Finally, transfer begins; 10 MB @ 50 MB/s leads to 1/5th of a second, or 200 ms, spent in transfer. Thus, for each 10 MB request, we spend 210 ms completing the request. To compute S, we just need to divide:

$$S = \frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ MB}{210\ ms} = 47.62\ MB/s$$

As we can see, because of the large time spent transferring data, S is very near the peak bandwidth of the disk (the seek and rotational costs have been amortized).

We can compute R similarly. Seek and rotation are the same; we then compute the time spent in transfer, which is 10 KB @ 50 MB/s, or 0.195 ms.

$$R = \frac{Amount\ of\ Data}{Time\ to\ access} = \frac{10\ KB}{10.195\ ms} = 0.981\ MB/s$$

As we can see, R is less than 1 MB/s, and S/R is almost 50.

Back To RAID-0 Analysis, Again

Let's now evaluate the performance of striping. As we said above, it is generally good. From a latency perspective, for example, the latency of a single-block request should be just about identical to that of a single disk; after all, RAID-0 will simply redirect that request to one of its disks.

From the perspective of steady-state sequential throughput, we'd expect to get the full bandwidth of the system. Thus, throughput equals N (the number of disks) multiplied by S (the sequential bandwidth of a single disk). For a large number of random I/Os, we can again use all of the disks, and thus obtain $N \cdot R$ MB/s. As we will see below, these values are both the simplest to calculate and will serve as an upper bound in comparison with other RAID levels.

38.5 RAID Level 1: Mirroring

Our first RAID level beyond striping is known as RAID level 1, or mirroring. With a mirrored system, we simply make more than one copy of each block in the system; each copy should be placed on a separate disk, of course. By doing so, we can tolerate disk failures.

In a typical mirrored system, we will assume that for each logical block, the RAID keeps two physical copies of it. Here is an example:

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: **Simple RAID-1: Mirroring**

In the example, disk 0 and disk 1 have identical contents, and disk 2 and disk 3 do as well; the data is striped across these mirror pairs. In fact, you may have noticed that there are a number of different ways to place block copies across the disks. The arrangement above is a common one and is sometimes called **RAID-10** (or **RAID 1+0**, **stripe of mirrors**) because it uses mirrored pairs (RAID-1) and then stripes (RAID-0) on top of them; another common arrangement is **RAID-01** (or **RAID 0+1**, **mirror of stripes**), which contains two large striping (RAID-0) arrays, and then mirrors (RAID-1) on top of them. For now, we will just talk about mirroring assuming the above layout.

When reading a block from a mirrored array, the RAID has a choice: it can read either copy. For example, if a read to logical block 5 is issued to the RAID, it is free to read it from either disk 2 or disk 3. When writing a block, though, no such choice exists: the RAID must update *both* copies of the data, in order to preserve reliability. Do note, though, that these writes can take place in parallel; for example, a write to logical block 5 could proceed to disks 2 and 3 at the same time.

RAID-1 Analysis

Let us assess RAID-1. From a capacity standpoint, RAID-1 is expensive; with the mirroring level = 2, we only obtain half of our peak useful capacity. With N disks of B blocks, RAID-1 useful capacity is $(N \cdot B)/2$.

From a reliability standpoint, RAID-1 does well. It can tolerate the failure of any one disk. You may also notice RAID-1 can actually do better than this, with a little luck. Imagine, in the figure above, that disk 0 and disk 2 both failed. In such a situation, there is no data loss! More generally, a mirrored system (with mirroring level of 2) can tolerate 1 disk failure for certain, and up to N/2 failures depending on which disks fail. In practice, we generally don't like to leave things like this to chance; thus most people consider mirroring to be good for handling a single failure.

Finally, we analyze performance. From the perspective of the latency of a single read request, we can see it is the same as the latency on a single disk; all the RAID-1 does is direct the read to one of its copies. A write is a little different: it requires two physical writes to complete before it is done. These two writes happen in parallel, and thus the time will be roughly equivalent to the time of a single write; however, because the logical write must wait for both physical writes to complete, it suffers the

ASIDE: THE RAID CONSISTENT-UPDATE PROBLEM

Before analyzing RAID-1, let us first discuss a problem that arises in any multi-disk RAID system, known as the **consistent-update problem** [DAA05]. The problem occurs on a write to any RAID that has to update multiple disks during a single logical operation. In this case, let us assume we are considering a mirrored disk array.

Imagine the write is issued to the RAID, and then the RAID decides that it must be written to two disks, disk 0 and disk 1. The RAID then issues the write to disk 0, but just before the RAID can issue the request to disk 1, a power loss (or system crash) occurs. In this unfortunate case, let us assume that the request to disk 0 completed (but clearly the request to disk 1 did not, as it was never issued).

The result of this untimely power loss is that the two copies of the block are now **inconsistent**; the copy on disk 0 is the new version, and the copy on disk 1 is the old. What we would like to happen is for the state of both disks to change **atomically**, i.e., either both should end up as the new version or neither.

The general way to solve this problem is to use a **write-ahead log** of some kind to first record what the RAID is about to do (i.e., update two disks with a certain piece of data) before doing it. By taking this approach, we can ensure that in the presence of a crash, the right thing will happen; by running a **recovery** procedure that replays all pending transactions to the RAID, we can ensure that no two mirrored copies (in the RAID-1 case) are out of sync.

One last note: because logging to disk on every write is prohibitively expensive, most RAID hardware includes a small amount of non-volatile RAM (e.g., battery-backed) where it performs this type of logging. Thus, consistent update is provided without the high cost of logging to disk.

worst-case seek and rotational delay of the two requests, and thus (on average) will be slightly higher than a write to a single disk.

To analyze steady-state throughput, let us start with the sequential workload. When writing out to disk sequentially, each logical write must result in two physical writes; for example, when we write logical block 0 (in the figure above), the RAID internally would write it to both disk 0 and disk 1. Thus, we can conclude that the maximum bandwidth obtained during sequential writing to a mirrored array is $(\frac{N}{2} \cdot S)$, or half the peak bandwidth.

Unfortunately, we obtain the exact same performance during a sequential read. One might think that a sequential read could do better, because it only needs to read one copy of the data, not both. However, let's use an example to illustrate why this doesn't help much. Imagine we need to read blocks 0, 1, 2, 3, 4, 5, 6, and 7. Let's say we issue the read of 0 to disk 0, the read of 1 to disk 2, the read of 2 to disk 1, and the read of 3 to disk 3. We continue by issuing reads to 4, 5, 6, and 7 to disks 0, 2, 1,

and 3, respectively. One might naively think that because we are utilizing all disks, we are achieving the full bandwidth of the array.

To see that this is not (necessarily) the case, however, consider the requests a single disk receives (say disk 0). First, it gets a request for block 0; then, it gets a request for block 4 (skipping block 2). In fact, each disk receives a request for every other block. While it is rotating over the skipped block, it is not delivering useful bandwidth to the client. Thus, each disk will only deliver half its peak bandwidth. And thus, the sequential read will only obtain a bandwidth of ($\frac{N}{2} \cdot S$) MB/s.

Random reads are the best case for a mirrored RAID. In this case, we can distribute the reads across all the disks, and thus obtain the full possible bandwidth. Thus, for random reads, RAID-1 delivers $N \cdot R$ MB/s.

Finally, random writes perform as you might expect: $\frac{N}{2} \cdot R$ MB/s. Each logical write must turn into two physical writes, and thus while all the disks will be in use, the client will only perceive this as half the available bandwidth. Even though a write to logical block x turns into two parallel writes to two different physical disks, the bandwidth of many small requests only achieves half of what we saw with striping. As we will soon see, getting half the available bandwidth is actually pretty good!

38.6 RAID Level 4: Saving Space With Parity

We now present a different method of adding redundancy to a disk array known as **parity**. Parity-based approaches attempt to use less capacity and thus overcome the huge space penalty paid by mirrored systems. They do so at a cost, however: performance.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.4: **RAID-4 With Parity**

Here is an example five-disk RAID-4 system (Figure 38.4). For each stripe of data, we have added a single **parity** block that stores the redundant information for that stripe of blocks. For example, parity block P1 has redundant information that it calculated from blocks 4, 5, 6, and 7.

To compute parity, we need to use a mathematical function that enables us to withstand the loss of any one block from our stripe. It turns out the simple function **XOR** does the trick quite nicely. For a given set of bits, the XOR of all of those bits returns a 0 if there are an even number of 1's in the bits, and a 1 if there are an odd number of 1's. For example:

In the first row (0,0,1,1), there are two 1's (C2, C3), and thus XOR of all of those values will be 0 (P); similarly, in the second row there is only

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

one 1 (C1), and thus the XOR must be 1 (P). You can remember this in a simple way: that the number of 1s in any row, including the parity bit, must be an even (not odd) number; that is the **invariant** that the RAID must maintain in order for parity to be correct.

From the example above, you might also be able to guess how parity information can be used to recover from a failure. Imagine the column labeled C2 is lost. To figure out what values must have been in the column, we simply have to read in all the other values in that row (including the XOR'd parity bit) and **reconstruct** the right answer. Specifically, assume the first row's value in column C2 is lost (it is a 1); by reading the other values in that row (0 from C0, 0 from C1, 1 from C3, and 0 from the parity column P), we get the values 0, 0, 1, and 0. Because we know that XOR keeps an even number of 1's in each row, we know what the missing data must be: a 1. And that is how reconstruction works in a XOR-based parity scheme! Note also how we compute the reconstructed value: we just XOR the data bits and the parity bits together, in the same way that we calculated the parity in the first place.

Now you might be wondering: we are talking about XORing all of these bits, and yet from above we know that the RAID places 4KB (or larger) blocks on each disk; how do we apply XOR to a bunch of blocks to compute the parity? It turns out this is easy as well. Simply perform a bitwise XOR across each bit of the data blocks; put the result of each bitwise XOR into the corresponding bit slot in the parity block. For example, if we had blocks of size 4 bits (yes, this is still quite a bit smaller than a 4KB block, but you get the picture), they might look something like this:

Block0	Block1	Block2	Block3	Parity
00	10	11	10	11
10	01	00	01	10

As you can see from the figure, the parity is computed for each bit of each block and the result placed in the parity block.

RAID-4 Analysis

Let us now analyze RAID-4. From a capacity standpoint, RAID-4 uses 1 disk for parity information for every group of disks it is protecting. Thus, our useful capacity for a RAID group is $(N - 1) \cdot B$.

Reliability is also quite easy to understand: RAID-4 tolerates 1 disk failure and no more. If more than one disk is lost, there is simply no way to reconstruct the lost data.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.5: **Full-stripe Writes In RAID-4**

Finally, there is performance. This time, let us start by analyzing steady-state throughput. Sequential read performance can utilize all of the disks except for the parity disk, and thus deliver a peak effective bandwidth of $(N-1)\cdot S$ MB/s (an easy case).

To understand the performance of sequential writes, we must first understand how they are done. When writing a big chunk of data to disk, RAID-4 can perform a simple optimization known as a **full-stripe write**. For example, imagine the case where the blocks 0, 1, 2, and 3 have been sent to the RAID as part of a write request (Figure 38.5).

In this case, the RAID can simply calculate the new value of P0 (by performing an XOR across the blocks 0, 1, 2, and 3) and then write all of the blocks (including the parity block) to the five disks above in parallel (highlighted in gray in the figure). Thus, full-stripe writes are the most efficient way for RAID-4 to write to disk.

Once we understand the full-stripe write, calculating the performance of sequential writes on RAID-4 is easy; the effective bandwidth is also $(N-1)\cdot S$ MB/s. Even though the parity disk is constantly in use during the operation, the client does not gain performance advantage from it.

Now let us analyze the performance of random reads. As you can also see from the figure above, a set of 1-block random reads will be spread across the data disks of the system but not the parity disk. Thus, the effective performance is: $(N-1)\cdot R$ MB/s.

Random writes, which we have saved for last, present the most interesting case for RAID-4. Imagine we wish to overwrite block 1 in the example above. We could just go ahead and overwrite it, but that would leave us with a problem: the parity block P0 would no longer accurately reflect the correct parity value of the stripe; in this example, P0 must also be updated. How can we update it both correctly and efficiently?

It turns out there are two methods. The first, known as **additive parity**, requires us to do the following. To compute the value of the new parity block, read in all of the other data blocks in the stripe in parallel (in the example, blocks 0, 2, and 3) and XOR those with the new block (1). The result is your new parity block. To complete the write, you can then write the new data and new parity to their respective disks, also in parallel.

The problem with this technique is that it scales with the number of disks, and thus in larger RAIDs requires a high number of reads to compute parity. Thus, the **subtractive parity** method.

For example, imagine this string of bits (4 data bits, one parity):

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0

Let's imagine that we wish to overwrite bit C2 with a new value which we will call $C2_{new}$. The subtractive method works in three steps. First, we read in the old data at C2 ($C2_{old}$ = 1) and the old parity (P_{old} = 0). Then, we compare the old data and the new data; if they are the same (e.g., $C2_{new}$ = $C2_{old}$), then we know the parity bit will also remain the same (i.e., P_{new} = P_{old}). If, however, they are different, then we must flip the old parity bit to the opposite of its current state, that is, if (P_{old} == 1), P_{new} will be set to 0; if (P_{old} == 0), P_{new} will be set to 1. We can express this whole mess neatly with XOR (where $\oplus$ is the XOR operator):

$$P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old} \tag{38.1}$$

Because we are dealing with blocks, not bits, we perform this calculation over all the bits in the block (e.g., 4096 bytes in each block multiplied by 8 bits per byte). Thus, in most cases, the new block will be different than the old block and thus the new parity block will too.

You should now be able to figure out when we would use the additive parity calculation and when we would use the subtractive method. Think about how many disks would need to be in the system so that the additive method performs fewer I/Os than the subtractive method; what is the cross-over point?

For this performance analysis, let us assume we are using the subtractive method. Thus, for each write, the RAID has to perform 4 physical I/Os (two reads and two writes). Now imagine there are lots of writes submitted to the RAID; how many can RAID-4 perform in parallel? To understand, let us again look at the RAID-4 layout (Figure 38.6).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Figure 38.6: **Example: Writes To 4, 13, And Respective Parity Blocks**

Now imagine there were 2 small writes submitted to the RAID-4 at about the same time, to blocks 4 and 13 (marked with * in the diagram). The data for those disks is on disks 0 and 1, and thus the read and write to data could happen in parallel, which is good. The problem that arises is with the parity disk; both the requests have to read the related parity blocks for 4 and 13, parity blocks 1 and 3 (marked with +). Hopefully, the issue is now clear: the parity disk is a bottleneck under this type of workload; we sometimes thus call this the **small-write problem** for parity-based RAIDs. Thus, even though the data disks could be accessed in

parallel, the parity disk prevents any parallelism from materializing; all writes to the system will be serialized because of the parity disk. Because the parity disk has to perform two I/Os (one read, one write) per logical I/O, we can compute the performance of small random writes in RAID-4 by computing the parity disk's performance on those two I/Os, and thus we achieve $(R/2)$ MB/s. RAID-4 throughput under random small writes is terrible; it does not improve as you add disks to the system.

We conclude by analyzing I/O latency in RAID-4. As you now know, a single read (assuming no failure) is just mapped to a single disk, and thus its latency is equivalent to the latency of a single disk request. The latency of a single write requires two reads and then two writes; the reads can happen in parallel, as can the writes, and thus total latency is about twice that of a single disk (with some differences because we have to wait for both reads to complete and thus get the worst-case positioning time, but then the updates don't incur seek cost and thus may be a better-than-average positioning cost).

38.7 RAID Level 5: Rotating Parity

To address the small-write problem (at least, partially), Patterson, Gibson, and Katz introduced RAID-5. RAID-5 works almost identically to RAID-4, except that it **rotates** the parity block across drives (Figure 38.7).

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figure 38.7: **RAID-5 With Rotated Parity**

As you can see, the parity block for each stripe is now rotated across the disks, in order to remove the parity-disk bottleneck for RAID-4.

RAID-5 Analysis

Much of the analysis for RAID-5 is identical to RAID-4. For example, the effective capacity and failure tolerance of the two levels are identical. So are sequential read and write performance. The latency of a single request (whether a read or a write) is also the same as RAID-4.

Random read performance is a little better, because we can now utilize all disks. Finally, random write performance improves noticeably over RAID-4, as it allows for parallelism across requests. Imagine a write to block 1 and a write to block 10; this will turn into requests to disk 1 and disk 4 (for block 1 and its parity) and requests to disk 0 and disk 2 (for

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S^1$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S^1$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

Figure 38.8: **RAID Capacity, Reliability, and Performance**

block 10 and its parity). Thus, they can proceed in parallel. In fact, we can generally assume that given a large number of random requests, we will be able to keep all the disks about evenly busy. If that is the case, then our total bandwidth for small writes will be $\frac{N}{4} \cdot R$ MB/s. The factor of four loss is due to the fact that each RAID-5 write still generates 4 total I/O operations, which is simply the cost of using parity-based RAID.

Because RAID-5 is basically identical to RAID-4 except in the few cases where it is better, it has almost completely replaced RAID-4 in the marketplace. The only place where it has not is in systems that know they will never perform anything other than a large write, thus avoiding the small-write problem altogether [HLM94]; in those cases, RAID-4 is sometimes used as it is slightly simpler to build.

38.8 RAID Comparison: A Summary

We now summarize our simplified comparison of RAID levels in Figure 38.8. Note that we have omitted a number of details to simplify our analysis. For example, when writing in a mirrored system, the average seek time is a little higher than when writing to just a single disk, because the seek time is the max of two seeks (one on each disk). Thus, random write performance to two disks will generally be a little less than random write performance of a single disk. Also, when updating the parity disk in RAID-4/5, the first read of the old parity will likely cause a full seek and rotation, but the second write of the parity will only result in rotation. Finally, sequential I/O to mirrored RAIDs pay a $2\times$ performance penalty as compared to other approaches[1].

[1]The $1/2$ penalty assumes a naive read/write pattern for mirroring; a more sophisticated approach that issued large I/O requests to differing parts of each mirror could potentially achieve full bandwidth. Think about this to see if you can figure out why.

However, the comparison in Figure 38.8 does capture the essential differences, and is useful for understanding tradeoffs across RAID levels. For the latency analysis, we simply use T to represent the time that a request to a single disk would take.

To conclude, if you strictly want performance and do not care about reliability, striping is obviously best. If, however, you want random I/O performance and reliability, mirroring is the best; the cost you pay is in lost capacity. If capacity and reliability are your main goals, then RAID-5 is the winner; the cost you pay is in small-write performance. Finally, if you are always doing sequential I/O and want to maximize capacity, RAID-5 also makes the most sense.

38.9 Other Interesting RAID Issues

There are a number of other interesting ideas that one could (and perhaps should) discuss when thinking about RAID. Here are some things we might eventually write about.

For example, there are many other RAID designs, including Levels 2 and 3 from the original taxonomy, and Level 6 to tolerate multiple disk faults [C+04]. There is also what the RAID does when a disk fails; sometimes it has a **hot spare** sitting around to fill in for the failed disk. What happens to performance under failure, and performance during reconstruction of the failed disk? There are also more realistic fault models, to take into account **latent sector errors** or **block corruption** [B+08], and lots of techniques to handle such faults (see the data integrity chapter for details). Finally, you can even build RAID as a software layer: such **software RAID** systems are cheaper but have other problems, including the consistent-update problem [DAA05].

38.10 Summary

We have discussed RAID. RAID transforms a number of independent disks into a large, more capacious, and more reliable single entity; importantly, it does so transparently, and thus hardware and software above is relatively oblivious to the change.

There are many possible RAID levels to choose from, and the exact RAID level to use depends heavily on what is important to the end-user. For example, mirrored RAID is simple, reliable, and generally provides good performance but at a high capacity cost. RAID-5, in contrast, is reliable and better from a capacity standpoint, but performs quite poorly when there are small writes in the workload. Picking a RAID and setting its parameters (chunk size, number of disks, etc.) properly for a particular workload is challenging, and remains more of an art than a science.

References

[B+08] "An Analysis of Data Corruption in the Storage Stack" by Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '08, San Jose, CA, February 2008. *Our own work analyzing how often disks actually corrupt your data. Not often, but sometimes! And thus something a reliable storage system must consider.*

[BJ88] "Disk Shadowing" by D. Bitton and J. Gray. VLDB 1988. *One of the first papers to discuss mirroring, therein called "shadowing".*

[CL95] "Striping in a RAID level 5 disk array" by Peter M. Chen and Edward K. Lee. SIGMETRICS 1995. *A nice analysis of some of the important parameters in a RAID-5 disk array.*

[C+04] "Row-Diagonal Parity for Double Disk Failure Correction" by P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, S. Sankar. FAST '04, February 2004. *Though not the first paper on a RAID system with two disks for parity, it is a recent and highly-understandable version of said idea. Read it to learn more.*

[DAA05] "Journal-guided Resynchronization for Software RAID" by Timothy E. Denehy, A. Arpaci-Dusseau, R. Arpaci-Dusseau. FAST 2005. *Our own work on the consistent-update problem. Here we solve it for Software RAID by integrating the journaling machinery of the file system above with the software RAID beneath it.*

[HLM94] "File System Design for an NFS File Server Appliance" by Dave Hitz, James Lau, Michael Malcolm. USENIX Winter 1994, San Francisco, California, 1994. *The sparse paper introducing a landmark product in storage, the write-anywhere file layout or WAFL file system that underlies the NetApp file server.*

[K86] "Synchronized Disk Interleaving" by M.Y. Kim. IEEE Transactions on Computers, Volume C-35: 11, November 1986. *Some of the earliest work on RAID is found here.*

[K88] "Small Disk Arrays – The Emerging Approach to High Performance" by F. Kurzweil. Presentation at Spring COMPCON '88, March 1, 1988, San Francisco, California. *Another early RAID reference.*

[P+88] "Redundant Arrays of Inexpensive Disks" by D. Patterson, G. Gibson, R. Katz. SIGMOD 1988. *This is considered **the** RAID paper, written by famous authors Patterson, Gibson, and Katz. The paper has since won many test-of-time awards and ushered in the RAID era, including the name RAID itself!*

[PB86] "Providing Fault Tolerance in Parallel Secondary Storage Systems" by A. Park, K. Balasubramaniam. Department of Computer Science, Princeton, CS-TR-O57-86, November 1986. *Another early work on RAID.*

[SG86] "Disk Striping" by K. Salem, H. Garcia-Molina. IEEE International Conference on Data Engineering, 1986. *And yes, another early RAID work. There are a lot of these, which kind of came out of the woodwork when the RAID paper was published in SIGMOD.*

[S84] "Byzantine Generals in Action: Implementing Fail-Stop Processors" by F.B. Schneider. ACM Transactions on Computer Systems, 2(2):145–154, May 1984. *Finally, a paper that is not about RAID! This paper is actually about how systems fail, and how to make something behave in a fail-stop manner.*

Homework (Simulation)

This section introduces `raid.py`, a simple RAID simulator you can use to shore up your knowledge of how RAID systems work. See the README for details.

Questions

1. Use the simulator to perform some basic RAID mapping tests. Run with different levels (0, 1, 4, 5) and see if you can figure out the mappings of a set of requests. For RAID-5, see if you can figure out the difference between left-symmetric and left-asymmetric layouts. Use some different random seeds to generate different problems than above.
2. Do the same as the first problem, but this time vary the chunk size with `-C`. How does chunk size change the mappings?
3. Do the same as above, but use the `-r` flag to reverse the nature of each problem.
4. Now use the reverse flag but increase the size of each request with the `-S` flag. Try specifying sizes of 8k, 12k, and 16k, while varying the RAID level. What happens to the underlying I/O pattern when the size of the request increases? Make sure to try this with the sequential workload too (`-W sequential`); for what request sizes are RAID-4 and RAID-5 much more I/O efficient?
5. Use the timing mode of the simulator (`-t`) to estimate the performance of 100 random reads to the RAID, while varying the RAID levels, using 4 disks.
6. Do the same as above, but increase the number of disks. How does the performance of each RAID level scale as the number of disks increases?
7. Do the same as above, but use all writes (`-w 100`) instead of reads. How does the performance of each RAID level scale now? Can you do a rough estimate of the time it will take to complete the workload of 100 random writes?
8. Run the timing mode one last time, but this time with a sequential workload (`-W sequential`). How does the performance vary with RAID level, and when doing reads versus writes? How about when varying the size of each request? What size should you write to a RAID when using RAID-4 or RAID-5?

39

Interlude: Files and Directories

Thus far we have seen the development of two key operating system abstractions: the process, which is a virtualization of the CPU, and the address space, which is a virtualization of memory. In tandem, these two abstractions allow a program to run as if it is in its own private, isolated world; as if it has its own processor (or processors); as if it has its own memory. This illusion makes programming the system much easier and thus is prevalent today not only on desktops and servers but increasingly on all programmable platforms including mobile phones and the like.

In this section, we add one more critical piece to the virtualization puzzle: **persistent storage**. A persistent-storage device, such as a classic **hard disk drive** or a more modern **solid-state storage device**, stores information permanently (or at least, for a long time). Unlike memory, whose contents are lost when there is a power loss, a persistent-storage device keeps such data intact. Thus, the OS must take extra care with such a device: this is where users keep data that they really care about.

CRUX: HOW TO MANAGE A PERSISTENT DEVICE
How should the OS manage a persistent device? What are the APIs? What are the important aspects of the implementation?

Thus, in the next few chapters, we will explore critical techniques for managing persistent data, focusing on methods to improve performance and reliability. We begin, however, with an overview of the API: the interfaces you'll expect to see when interacting with a UNIX file system.

39.1 Files And Directories

Two key abstractions have developed over time in the virtualization of storage. The first is the **file**. A file is simply a linear array of bytes, each of which you can read or write. Each file has some kind of **low-level name**, usually a number of some kind; often, the user is not aware of this

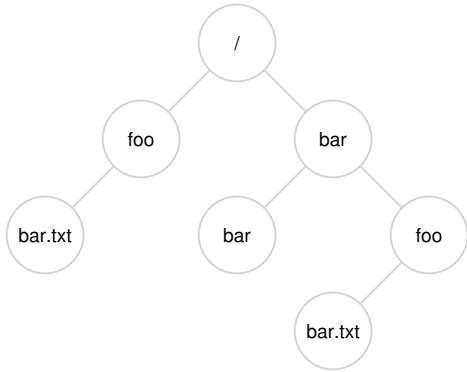


Figure 39.1: **An Example Directory Tree**

name (as we will see). For historical reasons, the low-level name of a file is often referred to as its **inode number** (**i-number**). We'll be learning a lot more about inodes in future chapters; for now, just assume that each file has an inode number associated with it.

In most systems, the OS does not know much about the structure of the file (e.g., whether it is a picture, or a text file, or C code); rather, the responsibility of the file system is simply to store such data persistently on disk and make sure that when you request the data again, you get what you put there in the first place. Doing so is not as simple as it seems!

The second abstraction is that of a **directory**. A directory, like a file, also has a low-level name (i.e., an inode number), but its contents are quite specific: it contains a list of (user-readable name, low-level name) pairs. For example, let's say there is a file with the low-level name "10", and it is referred to by the user-readable name of "foo". The directory that "foo" resides in thus would have an entry ("foo", "10") that maps the user-readable name to the low-level name. Each entry in a directory refers to either files or other directories. By placing directories within other directories, users are able to build an arbitrary **directory tree** (or **directory hierarchy**), under which all files and directories are stored.

The directory hierarchy starts at a **root directory** (in UNIX-based systems, the root directory is simply referred to as `/`) and uses some kind of **separator** to name subsequent **sub-directories** until the desired file or directory is named. For example, if a user created a directory `foo` in the root directory `/`, and then created a file `bar.txt` in the directory `foo`, we could refer to the file by its **absolute pathname**, which in this case would be `/foo/bar.txt`. See Figure 39.1 for a more complex directory tree; valid directories in the example are `/`, `/foo`, `/bar`, `/bar/bar`, `/bar/foo` and valid files are `/foo/bar.txt` and `/bar/foo/bar.txt`.

TIP: THINK CAREFULLY ABOUT NAMING
Naming is an important aspect of computer systems [SK09]. In UNIX systems, virtually everything that you can think of is named through the file system. Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file system. This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

Directories and files can have the same name as long as they are in different locations in the file-system tree (e.g., there are two files named `bar.txt` in the figure, `/foo/bar.txt` and `/bar/foo/bar.txt`).

You may also notice that the file name in this example often has two parts: `bar` and `txt`, separated by a period. The first part is an arbitrary name, whereas the second part of the file name is usually used to indicate the **type** of the file, e.g., whether it is C code (e.g., `.c`), or an image (e.g., `.jpg`), or a music file (e.g., `.mp3`). However, this is usually just a **convention**: there is usually no enforcement that the data contained in a file named `main.c` is indeed C source code.

Thus, we can see one great thing provided by the file system: a convenient way to **name** all the files we are interested in. Names are important in systems as the first step to accessing any resource is being able to name it. In UNIX systems, the file system thus provides a unified way to access files on disk, USB stick, CD-ROM, many other devices, and in fact many other things, all located under the single directory tree.

39.2 The File System Interface

Let's now discuss the file system interface in more detail. We'll start with the basics of creating, accessing, and deleting files. You may think this is straightforward, but along the way we'll discover the mysterious call that is used to remove files, known as `unlink()`. Hopefully, by the end of this chapter, this mystery won't be so mysterious to you!

39.3 Creating Files

We'll start with the most basic of operations: creating a file. This can be accomplished with the `open` system call; by calling `open()` and passing it the `O_CREAT` flag, a program can create a new file. Here is some example code to create a file called "foo" in the current working directory:

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
```

ASIDE: THE `CREAT()` SYSTEM CALL

The older way of creating a file is to call `creat()`, as follows:

```
// option: add second flag to set permissions
int fd = creat("foo");
```

You can think of `creat()` as `open()` with the following flags: `O_CREAT | O_WRONLY | O_TRUNC`. Because `open()` can create a file, the usage of `creat()` has somewhat fallen out of favor (indeed, it could just be implemented as a library call to `open()`); however, it does hold a special place in UNIX lore. Specifically, when Ken Thompson was asked what he would do differently if he were redesigning UNIX, he replied: "I'd spell creat with an e."

The routine `open()` takes a number of different flags. In this example, the second parameter creates the file (`O_CREAT`) if it does not exist, ensures that the file can only be written to (`O_WRONLY`), and, if the file already exists, truncates it to a size of zero bytes thus removing any existing content (`O_TRUNC`). The third parameter specifies permissions, in this case making the file readable and writable by the owner.

One important aspect of `open()` is what it returns: a **file descriptor**. A file descriptor is just an integer, private per process, and is used in UNIX systems to access files; thus, once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so. In this way, a file descriptor is a **capability** [L84], i.e., an opaque handle that gives you the power to perform certain operations. Another way to think of a file descriptor is as a pointer to an object of type file; once you have such an object, you can call other "methods" to access the file, like `read()` and `write()` (we'll see how to do so below).

As stated above, file descriptors are managed by the operating system on a per-process basis. This means some kind of simple structure (e.g., an array) is kept in the `proc` structure on UNIX systems. Here is the relevant piece from the xv6 kernel [CK+08]:

```
struct proc {
  ...
  struct file *ofile[NOFILE]; // Open files
  ...
};
```

A simple array (with a maximum of `NOFILE` open files), indexed by the file descriptor, tracks which files are opened on a per-process basis. Each entry of the array is actually just a pointer to a `struct file`, which will be used to track information about the file being read or written; we'll discuss this further below.

TIP: USE STRACE (AND SIMILAR TOOLS)
The `strace` tool provides an awesome way to see what programs are up to. By running it, you can trace which system calls a program makes, see the arguments and return codes, and generally get a very good idea of what is going on.
The tool also takes some arguments which can be quite useful. For example, `-f` follows any fork'd children too; `-t` reports the time of day at each call; `-e trace=open,close,read,write` only traces calls to those system calls and ignores all others. There are many other flags; read the man pages and find out how to harness this wonderful tool.

39.4 Reading And Writing Files

Once we have some files, of course we might like to read or write them. Let's start by reading an existing file. If we were typing at a command line, we might just use the program `cat` to dump the contents of the file to the screen.

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

In this code snippet, we redirect the output of the program `echo` to the file `foo`, which then contains the word "hello" in it. We then use `cat` to see the contents of the file. But how does the `cat` program access the file `foo`?

To find this out, we'll use an incredibly useful tool to trace the system calls made by a program. On Linux, the tool is called **strace**; other systems have similar tools (see **dtruss** on a Mac, or **truss** on some older UNIX variants). What `strace` does is trace every system call made by a program while it runs, and dump the trace to the screen for you to see.

Here is an example of using `strace` to figure out what `cat` is doing (some calls removed for readability):

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)       = 3
read(3, "hello\n", 4096)                = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                       = 0
close(3)                                = 0
...
prompt>
```

The first thing that `cat` does is open the file for reading. A couple of things we should note about this; first, that the file is only opened for reading (not writing), as indicated by the `O_RDONLY` flag; second, that the 64-bit offset is used (`O_LARGEFILE`); third, that the call to `open()` succeeds and returns a file descriptor, which has the value of 3.

Why does the first call to `open()` return 3, not 0 or perhaps 1 as you might expect? As it turns out, each running process already has three files open, standard input (which the process can read to receive input), standard output (which the process can write to in order to dump information to the screen), and standard error (which the process can write error messages to). These are represented by file descriptors 0, 1, and 2, respectively. Thus, when you first open another file (as `cat` does above), it will almost certainly be file descriptor 3.

After the open succeeds, `cat` uses the `read()` system call to repeatedly read some bytes from a file. The first argument to `read()` is the file descriptor, thus telling the file system which file to read; a process can of course have multiple files open at once, and thus the descriptor enables the operating system to know which file a particular read refers to. The second argument points to a buffer where the result of the `read()` will be placed; in the system-call trace above, strace shows the results of the read in this spot ("hello"). The third argument is the size of the buffer, which in this case is 4 KB. The call to `read()` returns successfully as well, here returning the number of bytes it read (6, which includes 5 for the letters in the word "hello" and one for an end-of-line marker).

At this point, you see another interesting result of the strace: a single call to the `write()` system call, to the file descriptor 1. As we mentioned above, this descriptor is known as the standard output, and thus is used to write the word "hello" to the screen as the program `cat` is meant to do. But does it call `write()` directly? Maybe (if it is highly optimized). But if not, what `cat` might do is call the library routine `printf()`; internally, `printf()` figures out all the formatting details passed to it, and eventually writes to standard output to print the results to the screen.

The `cat` program then tries to read more from the file, but since there are no bytes left in the file, the `read()` returns 0 and the program knows that this means it has read the entire file. Thus, the program calls `close()` to indicate that it is done with the file "foo", passing in the corresponding file descriptor. The file is thus closed, and the reading of it thus complete.

Writing a file is accomplished via a similar set of steps. First, a file is opened for writing, then the `write()` system call is called, perhaps repeatedly for larger files, and then `close()`. Use `strace` to trace writes to a file, perhaps of a program you wrote yourself, or by tracing the `dd` utility, e.g., `dd if=foo of=bar`.

ASIDE: DATA STRUCTURE — THE OPEN FILE TABLE
Each process maintains an array of file descriptors, each of which refers to an entry in the system-wide **open file table**. Each entry in this table tracks which underlying file the descriptor refers to, the current offset, and other relevant details such as whether the file is readable or writable.

39.5 Reading And Writing, But Not Sequentially

Thus far, we've discussed how to read and write files, but all access has been **sequential**; that is, we have either read a file from the beginning to the end, or written a file out from beginning to end.

Sometimes, however, it is useful to be able to read or write to a specific offset within a file; for example, if you build an index over a text document, and use it to look up a specific word, you may end up reading from some **random** offsets within the document. To do so, we will use the `lseek()` system call. Here is the function prototype:

```
off_t lseek(int fildes, off_t offset, int whence);
```

The first argument is familiar (a file descriptor). The second argument is the `offset`, which positions the **file offset** to a particular location within the file. The third argument, called `whence` for historical reasons, determines exactly how the seek is performed. From the man page:

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
  location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
  the file plus offset bytes.
```

As you can tell from this description, for each file a process opens, the OS tracks a "current" offset, which determines where the next read or write will begin reading from or writing to within the file. Thus, part of the abstraction of an open file is that it has a current offset, which is updated in one of two ways. The first is when a read or write of N bytes takes place, N is added to the current offset; thus each read or write *implicitly* updates the offset. The second is *explicitly* with `lseek`, which changes the offset as specified above.

The offset, as you might have guessed, is kept in that `struct file` we saw earlier, as referenced from the `struct proc`. Here is a (simplified) xv6 definition of the structure:

```
struct file {
  int ref;
  char readable;
  char writable;
  struct inode *ip;
  uint off;
};
```

ASIDE: CALLING LSEEK() DOES NOT PERFORM A DISK SEEK
The poorly-named system call `lseek()` confuses many a student trying to understand disks and how the file systems atop them work. Do not confuse the two! The `lseek()` call simply changes a variable in OS memory that tracks, for a particular process, at which offset its next read or write will start. A disk seek occurs when a read or write issued to the disk is not on the same track as the last read or write, and thus necessitates a head movement. Making this even more confusing is the fact that calling `lseek()` to read or write from/to random parts of a file, and then reading/writing to those random parts, will indeed lead to more disk seeks. Thus, calling `lseek()` can lead to a seek in an upcoming read or write, but absolutely does not cause any disk I/O to occur itself.

As you can see in the structure, the OS can use this to determine whether the opened file is readable or writable (or both), which underlying file it refers to (as pointed to by the `struct inode` pointer `ip`), and the current offset (`off`). There is also a reference count (`ref`), which we will discuss further below.

These file structures represent all of the currently opened files in the system; together, they are sometimes referred to as the **open file table**. The xv6 kernel just keeps these as an array, with one lock for the entire table:

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

Let's make this a bit clearer with a few examples. First, let's track a process that opens a file (of size 300 bytes) and reads it by calling the `read()` system call repeatedly, each time reading 100 bytes. Here is a trace of the relevant system calls, along with the values returned by each system call, and the value of the current offset in the open file table for this file access:

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

There are a couple of items of interest to note from the trace. First, you can see how the current offset gets initialized to zero when the file is

opened. Next, you can see how it is incremented with each `read()` by the process; this makes it easy for a process to just keep calling `read()` to get the next chunk of the file. Finally, you can see how at the end, an attempted `read()` past the end of the file returns zero, thus indicating to the process that it has read the file in its entirety.

Second, let's trace a process that opens the *same* file twice and issues a read to each of them.

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

In this example, two file descriptors are allocated (`3` and `4`), and each refers to a *different* entry in the open file table (in this example, entries `10` and `11`, as shown in the table heading; OFT stands for Open File Table). If you trace through what happens, you can see how each current offset is updated independently.

In one final example, a process uses `lseek()` to reposition the current offset before reading; in this case, only a single open file table entry is needed (as with the first example).

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

Here, the `lseek()` call first sets the current offset to `200`. The subsequent `read()` then reads the next 50 bytes, and updates the current offset accordingly.

39.6 Shared File Table Entries: `fork()` And `dup()`

In many cases (as in the examples shown above), the mapping of file descriptor to an entry in the open file table is a one-to-one mapping. For example, when a process runs, it might decide to open a file, read it, and then close it; in this example, the file will have a unique entry in the open file table. Even if some other process reads the same file at the same time, each will have its own entry in the open file table. In this way, each logical

```
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("child: offset %d\n", rc);
    } else if (rc > 0) {
        (void) wait(NULL);
        printf("parent: offset %d\n",
               (int) lseek(fd, 0, SEEK_CUR));
    }
    return 0;
}
```

Figure 39.2: **Shared Parent/Child File Table Entries (`fork-seek.c`)**

reading or writing of a file is independent, and each has its own current offset while it accesses the given file.

However, there are a few interesting cases where an entry in the open file table is *shared*. One of those cases occurs when a parent process creates a child process with `fork()`. Figure 39.2 shows a small code snippet in which a parent creates a child and then waits for it to complete. The child adjusts the current offset via a call to `lseek()` and then exits. Finally the parent, after waiting for the child, checks the current offset and prints out its value.

When we run this program, we see the following output:

```
prompt> ./fork-seek
child: offset 10
parent: offset 10
prompt>
```

Figure 39.3 shows the relationships that connect each process's private descriptor array, the shared open file table entry, and the reference from it to the underlying file-system inode. Note that we finally make use of the **reference count** here. When a file table entry is shared, its reference count is incremented; only when both processes close the file (or exit) will the entry be removed.

Sharing open file table entries across parent and child is occasionally useful. For example, if you create a number of processes that are cooperatively working on a task, they can write to the same output file without any extra coordination. For more on what is shared by processes when `fork()` is called, please see the man pages.

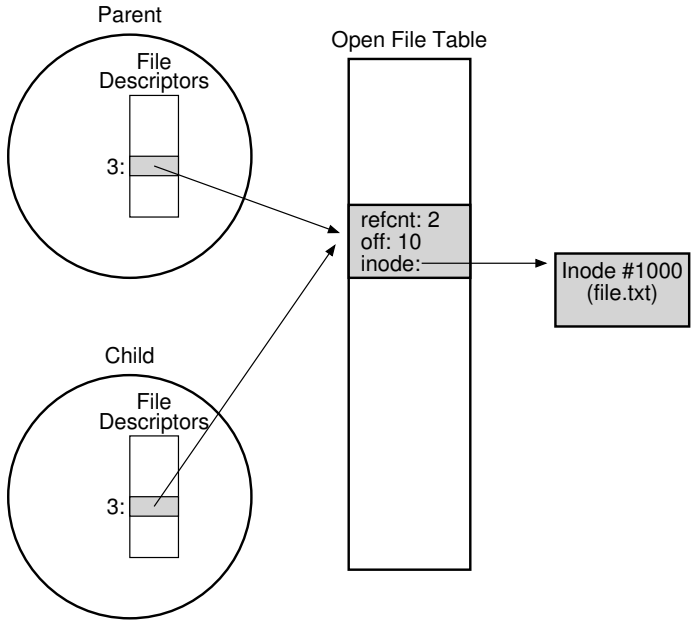


Figure 39.3: **Processes Sharing An Open File Table Entry**

One other interesting, and perhaps more useful, case of sharing occurs with the **dup()** system call (and its cousins, **dup2()** and **dup3()**).

The `dup()` call allows a process to create a new file descriptor that refers to the same underlying open file as an existing descriptor. Figure 39.4 shows a small code snippet that shows how `dup()` can be used.

The `dup()` call (and, in particular, `dup2()`) is useful when writing a UNIX shell and performing operations like output redirection; spend some time and think about why! And now, you are thinking: why didn't they tell me this when I was doing the shell project? Oh well, you can't get everything in the right order, even in an incredible book about operating systems. Sorry!

```
int main(int argc, char *argv[]) {
    int fd = open("README", O_RDONLY);
    assert(fd >= 0);
    int fd2 = dup(fd);
    // now fd and fd2 can be used interchangeably
    return 0;
}
```

Figure 39.4: **Shared File Table Entry With `dup()` (`dup.c`)**

39.7 Writing Immediately With `fsync()`

Most times when a program calls `write()`, it is just telling the file system: please write this data to persistent storage, at some point in the future. The file system, for performance reasons, will **buffer** such writes in memory for some time (say 5 seconds, or 30); at that later point in time, the write(s) will actually be issued to the storage device. From the perspective of the calling application, writes seem to complete quickly, and only in rare cases (e.g., the machine crashes after the `write()` call but before the write to disk) will data be lost.

However, some applications require something more than this eventual guarantee. For example, in a database management system (DBMS), development of a correct recovery protocol requires the ability to force writes to disk from time to time.

To support these types of applications, most file systems provide some additional control APIs. In the UNIX world, the interface provided to applications is known as `fsync(int fd)`. When a process calls `fsync()` for a particular file descriptor, the file system responds by forcing all **dirty** (i.e., not yet written) data to disk, for the file referred to by the specified file descriptor. The `fsync()` routine returns once all of these writes are complete.

Here is a simple example of how to use `fsync()`. The code opens the file `foo`, writes a single chunk of data to it, and then calls `fsync()` to ensure the writes are forced immediately to disk. Once the `fsync()` returns, the application can safely move on, knowing that the data has been persisted (if `fsync()` is correctly implemented, that is).

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

Interestingly, this sequence does not guarantee everything that you might expect; in some cases, you also need to `fsync()` the directory that contains the file `foo`. Adding this step ensures not only that the file itself is on disk, but that the file, if newly created, also is durably a part of the directory. Not surprisingly, this type of detail is often overlooked, leading to many application-level bugs [P+13,P+14].

39.8 Renaming Files

Once we have a file, it is sometimes useful to be able to give a file a different name. When typing at the command line, this is accomplished with `mv` command; in this example, the file `foo` is renamed `bar`:

ASIDE: MMAP() AND PERSISTENT MEMORY
(Guest Aside by Terence Kelly)

Memory mapping is an alternative way to access persistent data in files. The **mmap()** system call creates a correspondence between byte offsets in a file and virtual addresses in the calling process; the former is called the **backing file** and the latter its **in-memory image**. The process can then access the backing file using CPU instructions (i.e., loads and stores) to the in-memory image.

By combining the persistence of files with the access semantics of memory, file-backed memory mappings support a software abstraction called **persistent memory**. The persistent memory style of programming can streamline applications by eliminating translation between different data formats for memory and storage [K19].

```
1  p = mmap(NULL, file_size, PROT_READ|PROT_WRITE,
2           MAP_SHARED, fd, 0);
3  assert(p != MAP_FAILED);
4  for (int i = 1; i < argc; i++)
5    if (strcmp(argv[i], "pop") == 0) // pop
6      if (p->n > 0)  // stack not empty
7        printf("%d\n", p->stack[--p->n]);
8    } else {  // push
9      if (sizeof(pstack_t) + (1 + p->n) * sizeof(int)
10         <= file_size)  // stack not full
11       p->stack[p->n++] = atoi(argv[i]);
12   }
```

The program `pstack.c` (included on the OSTEP code github repo, with a snippet shown above) stores a persistent stack in file `ps.img`, which begins life as a bag of zeros, e.g., created on the command line via the `truncate` or `dd` utility. The file contains a count of the size of the stack and an array of integers holding stack contents. After `mmap()`-ing the backing file we can access the stack using C pointers to the in-memory image, e.g., `p->n` accesses the number of items on the stack, and `p->stack` the array of integers. Because the stack is persistent, data `push`'d by one invocation of `pstack` can be `pop`'d by the next.

A crash, e.g., between the increment and the assignment of the `push`, could leave our persistent stack in an inconsistent state. Applications prevent such damage by using mechanisms that update persistent memory atomically with respect to failure [K20].

```
prompt> mv foo bar
```

Using `strace`, we can see that `mv` uses the system call `rename(char *old, char *new)`, which takes precisely two arguments: the original name of the file (`old`) and the new name (`new`).

One interesting guarantee provided by the `rename()` call is that it is (usually) implemented as an **atomic** call with respect to system crashes; if the system crashes during the renaming, the file will either be named the old name or the new name, and no odd in-between state can arise. Thus, `rename()` is critical for supporting certain kinds of applications that require an atomic update to file state.

Let's be a little more specific here. Imagine that you are using a file editor (e.g., emacs), and you insert a line into the middle of a file. The file's name, for the example, is `foo.txt`. The way the editor might update the file to guarantee that the new file has the original contents plus the line inserted is as follows (ignoring error-checking for simplicity):

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

What the editor does in this example is simple: write out the new version of the file under a temporary name (`foo.txt.tmp`), force it to disk with `fsync()`, and then, when the application is certain the new file metadata and contents are on the disk, rename the temporary file to the original file's name. This last step atomically swaps the new file into place, while concurrently deleting the old version of the file, and thus an atomic file update is achieved.

39.9 Getting Information About Files

Beyond file access, we expect the file system to keep a fair amount of information about each file it is storing. We generally call such data about files **metadata**. To see the metadata for a certain file, we can use the `stat()` or `fstat()` system calls. These calls take a pathname (or file descriptor) to a file and fill in a `stat` structure as seen in Figure 39.5.

You can see that there is a lot of information kept about each file, including its size (in bytes), its low-level name (i.e., inode number), some ownership information, and some information about when the file was accessed or modified, among other things. To see this information, you can use the command line tool `stat`. In this example, we first create a file (called `file`) and then use the `stat` command line tool to learn some things about the file.

```
struct stat {
  dev_t     st_dev;     // ID of device containing file
  ino_t     st_ino;     // inode number
  mode_t    st_mode;    // protection
  nlink_t   st_nlink;   // number of hard links
  uid_t     st_uid;     // user ID of owner
  gid_t     st_gid;     // group ID of owner
  dev_t     st_rdev;    // device ID (if special file)
  off_t     st_size;    // total size, in bytes
  blksize_t st_blksize; // blocksize for filesystem I/O
  blkcnt_t  st_blocks;  // number of blocks allocated
  time_t    st_atime;   // time of last access
  time_t    st_mtime;   // time of last modification
  time_t    st_ctime;   // time of last status change
};
```

Figure 39.5: **The `stat` structure.**

Here is the output on Linux:

```
prompt> echo hello > file
prompt> stat file
  File: 'file'
  Size: 6   Blocks: 8   IO Block: 4096   regular file
Device: 811h/2065d Inode: 67158084    Links: 1
Access: (0640/-rw-r-----) Uid: (30686/remzi)
  Gid: (30686/remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

Each file system usually keeps this type of information in a structure called an **inode**[1]. We'll be learning a lot more about inodes when we talk about file system implementation. For now, you should just think of an inode as a persistent data structure kept by the file system that has information like we see above inside of it. All inodes reside on disk; a copy of active ones are usually cached in memory to speed up access.

39.10 Removing Files

At this point, we know how to create files and access them, either sequentially or not. But how do you delete files? If you've used UNIX, you probably think you know: just run the program `rm`. But what system call does `rm` use to remove a file?

[1]Some file systems call these structures similar, but slightly different, names, such as dnodes; the basic idea is similar however.

Let's use our old friend `strace` again to find out. Here we remove that pesky file `foo`:

```
prompt> strace rm foo
...
unlink("foo")                           = 0
...
```

We've removed a bunch of unrelated cruft from the traced output, leaving just a single call to the mysteriously-named system call `unlink()`. As you can see, `unlink()` just takes the name of the file to be removed, and returns zero upon success. But this leads us to a great puzzle: why is this system call named `unlink`? Why not just `remove` or `delete`? To understand the answer to this puzzle, we must first understand more than just files, but also directories.

39.11 Making Directories

Beyond files, a set of directory-related system calls enable you to make, read, and delete directories. Note you can never write to a directory directly. Because the format of the directory is considered file system metadata, the file system considers itself responsible for the integrity of directory data; thus, you can only update a directory indirectly by, for example, creating files, directories, or other object types within it. In this way, the file system makes sure that directory contents are as expected.

To create a directory, a single system call, `mkdir()`, is available. The eponymous `mkdir` program can be used to create such a directory. Let's take a look at what happens when we run the `mkdir` program to make a simple directory called `foo`:

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                      = 0
...
prompt>
```

When such a directory is created, it is considered "empty", although it does have a bare minimum of contents. Specifically, an empty directory has two entries: one entry that refers to itself, and one entry that refers to its parent. The former is referred to as the "." (dot) directory, and the latter as ".." (dot-dot). You can see these directories by passing a flag (`-a`) to the program `ls`:

```
prompt> ls -a
./  ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

TIP: BE WARY OF POWERFUL COMMANDS

The program `rm` provides us with a great example of powerful commands, and how sometimes too much power can be a bad thing. For example, to remove a bunch of files at once, you can type something like:

```
prompt> rm *
```

where the `*` will match all files in the current directory. But sometimes you want to also delete the directories too, and in fact all of their contents. You can do this by telling `rm` to recursively descend into each directory, and remove its contents too:

```
prompt> rm -rf *
```

Where you get into trouble with this small string of characters is when you issue the command, accidentally, from the root directory of a file system, thus removing every file and directory from it. Oops!

Thus, remember the double-edged sword of powerful commands; while they give you the ability to do a lot of work with a small number of keystrokes, they also can quickly and readily do a great deal of harm.

39.12 Reading Directories

Now that we've created a directory, we might wish to read one too. Indeed, that is exactly what the program `ls` does. Let's write our own little tool like `ls` and see how it is done.

Instead of just opening a directory as if it were a file, we instead use a new set of calls. Below is an example program that prints the contents of a directory. The program uses three calls, `opendir()`, `readdir()`, and `closedir()`, to get the job done, and you can see how simple the interface is; we just use a simple loop to read one directory entry at a time, and print out the name and inode number of each file in the directory.

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino,
               d->d_name);
    }
    closedir(dp);
    return 0;
}
```

The declaration below shows the information available within each directory entry in the `struct dirent` data structure:

```
struct dirent {
  char           d_name[256]; // filename
  ino_t          d_ino;       // inode number
  off_t          d_off;       // offset to the next dirent
  unsigned short d_reclen;    // length of this record
  unsigned char  d_type;      // type of file
};
```

Because directories are light on information (basically, just mapping the name to the inode number, along with a few other details), a program may want to call `stat()` on each file to get more information on each, such as its length or other detailed information. Indeed, this is exactly what `ls` does when you pass it the `-l` flag; try `strace` on `ls` with and without that flag to see for yourself.

39.13 Deleting Directories

Finally, you can delete a directory with a call to `rmdir()` (which is used by the program of the same name, `rmdir`). Unlike file deletion, however, removing directories is more dangerous, as you could potentially delete a large amount of data with a single command. Thus, `rmdir()` has the requirement that the directory be empty (i.e., only has "." and ".." entries) before it is deleted. If you try to delete a non-empty directory, the call to `rmdir()` simply will fail.

39.14 Hard Links

We now come back to the mystery of why removing a file is performed via `unlink()`, by understanding a new way to make an entry in the file system tree, through a system call known as `link()`. The `link()` system call takes two arguments, an old pathname and a new one; when you "link" a new file name to an old one, you essentially create another way to refer to the same file. The command-line program `ln` is used to do this, as we see in this example:

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

Here we created a file with the word "hello" in it, and called the file `file`[2]. We then create a hard link to that file using the `ln` program. After this, we can examine the file by either opening `file` or `file2`.

The way `link()` works is that it simply creates another name in the directory you are creating the link to, and refers it to the *same* inode number (i.e., low-level name) of the original file. The file is not copied in any way; rather, you now just have two human-readable names (`file` and `file2`) that both refer to the same file. We can even see this in the directory itself, by printing out the inode number of each file:

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

By passing the `-i` flag to `ls`, it prints out the inode number of each file (as well as the file name). And thus you can see what link really has done: just make a new reference to the same exact inode number (67158084 in this example).

By now you might be starting to see why `unlink()` is called `unlink()`. When you create a file, you are really doing *two* things. First, you are making a structure (the inode) that will track virtually all relevant information about the file, including its size, where its blocks are on disk, and so forth. Second, you are *linking* a human-readable name to that file, and putting that link into a directory.

After creating a hard link to a file, the file system perceives no difference between the original file name (`file`) and the newly created file name (`file2`); indeed, they are both just links to the underlying metadata about the file, which is found in inode number 67158084.

Thus, to remove a file from the file system, we call `unlink()`. In the example above, we could for example remove the file named `file`, and still access the file without difficulty:

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

The reason this works is because when the file system unlinks file, it checks a **reference count** within the inode number. This reference count (sometimes called the **link count**) allows the file system to track how many different file names have been linked to this particular inode. When `unlink()` is called, it removes the "link" between the human-readable

[2]Note again how creative the authors of this book are. We also used to have a cat named "Cat" (true story). However, she died, and we now have a hamster named "Hammy." Update: Hammy is now dead too. The pet bodies are piling up.

name (the file that is being deleted) to the given inode number, and decrements the reference count; only when the reference count reaches zero does the file system also free the inode and related data blocks, and thus truly "delete" the file.

You can see the reference count of a file using `stat()` of course. Let's see what it is when we create and delete hard links to a file. In this example, we'll create three links to the same file, and then delete them. Watch the link count!

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084    Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084    Links: 2 ...
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084    Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084    Links: 1 ...
prompt> rm file3
```

39.15 Symbolic Links

There is one other type of link that is really useful, and it is called a **symbolic link** or sometimes a **soft link**. Hard links are somewhat limited: you can't create one to a directory (for fear that you will create a cycle in the directory tree); you can't hard link to files in other disk partitions (because inode numbers are only unique within a particular file system, not across file systems); etc. Thus, a new type of link called the symbolic link was created [MJLF84].

To create such a link, you can use the same program `ln`, but with the `-s` flag. Here is an example:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

As you can see, creating a soft link looks much the same, and the original file can now be accessed through the file name `file` as well as the symbolic link name `file2`.

However, beyond this surface similarity, symbolic links are actually quite different from hard links. The first difference is that a symbolic link is actually a file itself, of a different type. We've already talked about regular files and directories; symbolic links are a third type the file system knows about. A `stat` on the symlink reveals all:

```
prompt> stat file
  ... regular file ...
prompt> stat file2
  ... symbolic link ...
```

Running `ls` also reveals this fact. If you look closely at the first character of the long-form of the output from `ls`, you can see that the first character in the left-most column is a `-` for regular files, a `d` for directories, and an `l` for soft links. You can also see the size of the symbolic link (4 bytes in this case) and what the link points to (the file named `file`).

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

The reason that `file2` is 4 bytes is because the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file. Because we've linked to a file named `file`, our link file `file2` is small (4 bytes). If we link to a longer pathname, our link file would be bigger:

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May  3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May  3 19:17 file3 ->
                                         alongerfilename
```

Finally, because of the way symbolic links are created, they leave the possibility for what is known as a **dangling reference**:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

As you can see in this example, quite unlike hard links, removing the original file named `file` causes the link to point to a pathname that no longer exists.

39.16 Permission Bits And Access Control Lists

The abstraction of a process provided two central virtualizations: of the CPU and of memory. Each of these gave the illusion to a process that it had its own *private* CPU and its own *private* memory; in reality, the OS underneath used various techniques to share limited physical resources among competing entities in a safe and secure manner.

The file system also presents a virtual view of a disk, transforming it from a bunch of raw blocks into much more user-friendly files and directories, as described within this chapter. However, the abstraction is notably different from that of the CPU and memory, in that files are commonly *shared* among different users and processes and are not (always) private. Thus, a more comprehensive set of mechanisms for enabling various degrees of sharing are usually present within file systems.

The first form of such mechanisms is the classic UNIX **permission bits**. To see permissions for a file `foo.txt`, just type:

```
prompt> ls -l foo.txt
-rw-r--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

We'll just pay attention to the first part of this output, namely the `-rw-r--r--`. The first character here just shows the type of the file: `-` for a regular file (which `foo.txt` is), `d` for a directory, `l` for a symbolic link, and so forth; this is (mostly) not related to permissions, so we'll ignore it for now.

We are interested in the permission bits, which are represented by the next nine characters (`rw-r--r--`). These bits determine, for each regular file, directory, and other entities, exactly who can access it and how.

The permissions consist of three groupings: what the **owner** of the file can do to it, what someone in a **group** can do to the file, and finally, what anyone (sometimes referred to as **other**) can do. The abilities the owner, group member, or others can have include the ability to read the file, write it, or execute it.

In the example above, the first three characters of the output of `ls` show that the file is both readable and writable by the owner (`rw-`), and only readable by members of the group `wheel` and also by anyone else in the system (`r--` followed by `r--`).

The owner of the file can readily change these permissions, for example by using the `chmod` command (to change the **file mode**). To remove the ability for anyone except the owner to access the file, you could type:

```
prompt> chmod 600 foo.txt
```

ASIDE: SUPERUSER FOR FILE SYSTEMS

Which user is allowed to do privileged operations to help administer the file system? For example, if an inactive user's files need to be deleted to save space, who has the rights to do so?

On local file systems, the common default is for there to be some kind of **superuser** (i.e., **root**) who can access all files regardless of privileges. In a distributed file system such as AFS (which has access control lists), a group called `system:administrators` contains users that are trusted to do so. In both cases, these trusted users represent an inherent security risk; if an attacker is able to somehow impersonate such a user, the attacker can access all the information in the system, thus violating expected privacy and protection guarantees.

This command enables the readable bit (4) and writable bit (2) for the owner (OR'ing them together yields the 6 above), but set the group and other permission bits to 0 and 0, respectively, thus setting the permissions to `rw-------`.

The execute bit is particularly interesting. For regular files, its presence determines whether a program can be run or not. For example, if we have a simple shell script called `hello.csh`, we may wish to run it by typing:

```
prompt> ./hello.csh
hello, from shell world.
```

However, if we don't set the execute bit properly for this file, the following happens:

```
prompt> chmod 600 hello.csh
prompt> ./hello.csh
./hello.csh: Permission denied.
```

For directories, the execute bit behaves a bit differently. Specifically, it enables a user (or group, or everyone) to do things like change directories (i.e., `cd`) into the given directory, and, in combination with the writable bit, create files therein. The best way to learn more about this: play around with it yourself! Don't worry, you (probably) won't mess anything up too badly.

Beyond permissions bits, some file systems, such as the distributed file system known as AFS (discussed in a later chapter), include more sophisticated controls. AFS, for example, does this in the form of an **access control list** (**ACL**) per directory. Access control lists are a more general and powerful way to represent exactly who can access a given resource. In a file system, this enables a user to create a very specific list of who can and cannot read a set of files, in contrast to the somewhat limited owner/group/everyone model of permissions bits described above.

For example, here are the access controls for a private directory in one author's AFS account, as shown by the `fs listacl` command:

```
prompt> fs listacl private
Access list for private is
Normal rights:
  system:administrators rlidwka
  remzi rlidwka
```

The listing shows that both the system administrators and the user `remzi` can lookup, insert, delete, and administer files in this directory, as well as read, write, and lock those files.

To allow someone (in this case, the other author) to access this directory, user `remzi` can just type the following command.

```
prompt> fs setacl private/ andrea rl
```

There goes `remzi`'s privacy! But now you have learned an even more important lesson: there can be no secrets in a good marriage, even within the file system[3].

39.17 Making And Mounting A File System

We've now toured the basic interfaces to access files, directories, and certain special types of links. But there is one more topic we should discuss: how to assemble a full directory tree from many underlying file systems. This task is accomplished via first making file systems, and then mounting them to make their contents accessible.

To make a file system, most file systems provide a tool, usually referred to as `mkfs` (pronounced "make fs"), that performs exactly this task. The idea is as follows: give the tool, as input, a device (such as a disk partition, e.g., `/dev/sda1`) and a file system type (e.g., ext3), and it simply writes an empty file system, starting with a root directory, onto that disk partition. And mkfs said, let there be a file system!

However, once such a file system is created, it needs to be made accessible within the uniform file-system tree. This task is achieved via the `mount` program (which makes the underlying system call `mount()` to do the real work). What mount does, quite simply is take an existing directory as a target **mount point** and essentially paste a new file system onto the directory tree at that point.

An example here might be useful. Imagine we have an unmounted ext3 file system, stored in device partition `/dev/sda1`, that has the following contents: a root directory which contains two sub-directories, `a` and `b`, each of which in turn holds a single file named `foo`. Let's say we wish to mount this file system at the mount point `/home/users`. We would type something like this:

[3] Married happily since 1996, if you were wondering. We know, you weren't.

TIP: BE WARY OF TOCTTOU

In 1974, McPhee noticed a problem in computer systems. Specifically, McPhee noted that "... if there exists a time interval between a validity-check and the operation connected with that validity-check, [and,] through multitasking, the validity-check variables can deliberately be changed during this time interval, resulting in an invalid operation being performed by the control program." We today call this the **Time Of Check To Time Of Use** (**TOCTTOU**) problem, and alas, it still can occur.

A simple example, as described by Bishop and Dilger [BD96], shows how a user can trick a more trusted service and thus cause trouble. Imagine, for example, that a mail service runs as root (and thus has privilege to access all files on a system). This service appends an incoming message to a user's inbox file as follows. First, it calls `lstat()` to get information about the file, specifically ensuring that it is actually just a regular file owned by the target user, and not a link to another file that the mail server should not be updating. Then, after the check succeeds, the server updates the file with the new message.

Unfortunately, the gap between the check and the update leads to a problem: the attacker (in this case, the user who is receiving the mail, and thus has permissions to access the inbox) switches the inbox file (via a call to `rename()`) to point to a sensitive file such as `/etc/passwd` (which holds information about users and their passwords). If this switch happens at just the right time (between the check and the access), the server will blithely update the sensitive file with the contents of the mail. The attacker can now write to the sensitive file by sending an email, an escalation in privilege; by updating `/etc/passwd`, the attacker can add an account with `root` privileges and thus gain control of the system.

There are not any simple and great solutions to the TOCTTOU problem [T+08]. One approach is to reduce the number of services that need root privileges to run, which helps. The `O_NOFOLLOW` flag makes it so that `open()` will fail if the target is a symbolic link, thus avoiding attacks that require said links. More radical approaches, such as using a **transactional file system** [H+18], would solve the problem, but there aren't many transactional file systems in wide deployment. Thus, the usual (lame) advice: be careful when you write code that runs with high privileges!

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

If successful, the mount would thus make this new file system available. However, note how the new file system is now accessed. To look at the contents of the root directory, we would use `ls` like this:

```
prompt> ls /home/users/
a b
```

As you can see, the pathname `/home/users/` now refers to the root of the newly-mounted directory. Similarly, we could access directories `a` and `b` with the pathnames `/home/users/a` and `/home/users/b`. Finally, the files named `foo` could be accessed via `/home/users/a/foo` and `/home/users/b/foo`. And thus the beauty of mount: instead of having a number of separate file systems, mount unifies all file systems into one tree, making naming uniform and convenient.

To see what is mounted on your system, and at which points, simply run the `mount` program. You'll see something like this:

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```

This crazy mix shows that a whole number of different file systems, including ext3 (a standard disk-based file system), the proc file system (a file system for accessing information about current processes), tmpfs (a file system just for temporary files), and AFS (a distributed file system) are all glued together onto this one machine's file-system tree.

39.18 Summary

The file system interface in UNIX systems (and indeed, in any system) is seemingly quite rudimentary, but there is a lot to understand if you wish to master it. Nothing is better, of course, than simply using it (a lot). So please do so! Of course, read more; as always, Stevens [SR05] is the place to begin.

ASIDE: KEY FILE SYSTEM TERMS

- A **file** is an array of bytes which can be created, read, written, and deleted. It has a low-level name (i.e., a number) that refers to it uniquely. The low-level name is often called an **i-number**.
- A **directory** is a collection of tuples, each of which contains a human-readable name and low-level name to which it maps. Each entry refers either to another directory or to a file. Each directory also has a low-level name (i-number) itself. A directory always has two special entries: the `.` entry, which refers to itself, and the `..` entry, which refers to its parent.
- A **directory tree** or **directory hierarchy** organizes all files and directories into a large tree, starting at the **root**.
- To access a file, a process must use a system call (usually, `open()`) to request permission from the operating system. If permission is granted, the OS returns a **file descriptor**, which can then be used for read or write access, as permissions and intent allow.
- Each file descriptor is a private, per-process entity, which refers to an entry in the **open file table**. The entry therein tracks which file this access refers to, the **current offset** of the file (i.e., which part of the file the next read or write will access), and other relevant information.
- Calls to `read()` and `write()` naturally update the current offset; otherwise, processes can use `lseek()` to change its value, enabling random access to different parts of the file.
- To force updates to persistent media, a process must use `fsync()` or related calls. However, doing so correctly while maintaining high performance is challenging [P+14], so think carefully when doing so.
- To have multiple human-readable names in the file system refer to the same underlying file, use **hard links** or **symbolic links**. Each is useful in different circumstances, so consider their strengths and weaknesses before usage. And remember, deleting a file is just performing that one last `unlink()` of it from the directory hierarchy.
- Most file systems have mechanisms to enable and disable sharing. A rudimentary form of such controls are provided by **permissions bits**; more sophisticated **access control lists** allow for more precise control over exactly who can access and manipulate information.

References

[BD96] "Checking for Race Conditions in File Accesses" by Matt Bishop, Michael Dilger. Computing Systems 9:2, 1996. *A great description of the TOCTTOU problem and its presence in file systems.*

[CK+08] "The xv6 Operating System" by Russ Cox, Frans Kaashoek, Robert Morris, Nickolai Zeldovich. From: https://github.com/mit-pdos/xv6-public. *As mentioned before, a cool and simple Unix implementation. We have been using an older version (2012-01-30-1-g1c41342) and hence some examples in the book may not match the latest in the source.*

[H+18] "TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions" by Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, E. Witchel. USENIX ATC '18, June 2018. *The best paper at USENIX ATC '18, and a good recent place to start to learn about transactional file systems.*

[K19] "Persistent Memory Programming on Conventional Hardware" by Terence Kelly. ACM Queue, 17:4, July/August 2019. *A great overview of persistent memory programming; check it out!*

[K20] "Is Persistent Memory Persistent?" by Terence Kelly. Communications of the ACM, 63:9, September 2020. *An engaging article about how to test hardware failures in system on the cheaps; who knew breaking things could be so fun?*

[K84] "Processes as Files" by Tom J. Killian. USENIX, June 1984. *The paper that introduced the /proc file system, where each process can be treated as a file within a pseudo file system. A clever idea that you can still see in modern* UNIX *systems.*

[L84] "Capability-Based Computer Systems" by Henry M. Levy. Digital Press, 1984. Available: http://homes.cs.washington.edu/~levy/capabook. *An excellent overview of early capability-based systems.*

[MJLF84] "A Fast File System for UNIX" by Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry. ACM TOCS, 2:3, August 1984. *We'll talk about the Fast File System (FFS) explicitly later on. Here, we refer to it because of all the other random fun things it introduced, like long file names and symbolic links. Sometimes, when you are building a system to improve one thing, you improve a lot of other things along the way.*

[P+13] "Towards Efficient, Portable Application-Level Consistency" by Thanumalayan S. Pillai, Vijay Chidambaram, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HotDep '13, November 2013. *Our own work that shows how readily applications can make mistakes in committing data to disk; in particular, assumptions about the file system creep into applications and thus make the applications work correctly only if they are running on a specific file system.*

[P+14] "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications" by Thanumalayan S. Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. OSDI '14, Broomfield, Colorado, October 2014. *The full conference paper on this topic – with many more details and interesting tidbits than the first workshop paper above.*

[SK09] "Principles of Computer System Design" by Jerome H. Saltzer and M. Frans Kaashoek. Morgan-Kaufmann, 2009. *This tour de force of systems is a must-read for anybody interested in the field. It's how they teach systems at MIT. Read it once, and then read it a few more times to let it all soak in.*

[SR05] "Advanced Programming in the UNIX Environment" by W. Richard Stevens and Stephen A. Rago. Addison-Wesley, 2005. *We have probably referenced this book a few hundred thousand times. It is that useful to you, if you care to become an awesome systems programmer.*

[T+08] "Portably Solving File TOCTTOU Races with Hardness Amplification" by D. Tsafrir, T. Hertz, D. Wagner, D. Da Silva. FAST '08, San Jose, California, 2008. *Not the paper that introduced TOCTTOU, but a recent-ish and well-done description of the problem and a way to solve the problem in a portable manner.*

Homework (Code)

In this homework, we'll just familiarize ourselves with how the APIs described in the chapter work. To do so, you'll just write a few different programs, mostly based on various UNIX utilities.

Questions

1. **Stat:** Write your own version of the command line program `stat`, which simply calls the `stat()` system call on a given file or directory. Print out file size, number of blocks allocated, reference (link) count, and so forth. What is the link count of a directory, as the number of entries in the directory changes? Useful interfaces: `stat()`, naturally.

2. **List Files:** Write a program that lists files in the given directory. When called without any arguments, the program should just print the file names. When invoked with the `-l` flag, the program should print out information about each file, such as the owner, group, permissions, and other information obtained from the `stat()` system call. The program should take one additional argument, which is the directory to read, e.g., `myls -l directory`. If no directory is given, the program should just use the current working directory. Useful interfaces: `stat()`, `opendir()`, `readdir()`, `getcwd()`.

3. **Tail:** Write a program that prints out the last few lines of a file. The program should be efficient, in that it seeks to near the end of the file, reads in a block of data, and then goes backwards until it finds the requested number of lines; at this point, it should print out those lines from beginning to the end of the file. To invoke the program, one should type: `mytail -n file`, where `n` is the number of lines at the end of the file to print. Useful interfaces: `stat()`, `lseek()`, `open()`, `read()`, `close()`.

4. **Recursive Search:** Write a program that prints out the names of each file and directory in the file system tree, starting at a given point in the tree. For example, when run without arguments, the program should start with the current working directory and print its contents, as well as the contents of any sub-directories, etc., until the entire tree, root at the CWD, is printed. If given a single argument (of a directory name), use that as the root of the tree instead. Refine your recursive search with more fun options, similar to the powerful `find` command line tool. Useful interfaces: figure it out.

40

File System Implementation

In this chapter, we introduce a simple file system implementation, known as **vsfs** (the **Very Simple File System**). This file system is a simplified version of a typical UNIX file system and thus serves to introduce some of the basic on-disk structures, access methods, and various policies that you will find in many file systems today.

The file system is pure software; unlike our development of CPU and memory virtualization, we will not be adding hardware features to make some aspect of the file system work better (though we will want to pay attention to device characteristics to make sure the file system works well). Because of the great flexibility we have in building a file system, many different ones have been built, literally from AFS (the Andrew File System) [H+88] to ZFS (Sun's Zettabyte File System) [B07]. All of these file systems have different data structures and do some things better or worse than their peers. Thus, the way we will be learning about file systems is through case studies: first, a simple file system (vsfs) in this chapter to introduce most concepts, and then a series of studies of real file systems to understand how they can differ in practice.

THE CRUX: HOW TO IMPLEMENT A SIMPLE FILE SYSTEM
How can we build a simple file system? What structures are needed on the disk? What do they need to track? How are they accessed?

40.1 The Way To Think

To think about file systems, we usually suggest thinking about two different aspects of them; if you understand both of these aspects, you probably understand how the file system basically works.

The first is the **data structures** of the file system. In other words, what types of on-disk structures are utilized by the file system to organize its data and metadata? The first file systems we'll see (including vsfs below) employ simple structures, like arrays of blocks or other objects, whereas

ASIDE: MENTAL MODELS OF FILE SYSTEMS
As we've discussed before, mental models are what you are really trying to develop when learning about systems. For file systems, your mental model should eventually include answers to questions like: what on-disk structures store the file system's data and metadata? What happens when a process opens a file? Which on-disk structures are accessed during a read or write? By working on and improving your mental model, you develop an abstract understanding of what is going on, instead of just trying to understand the specifics of some file-system code (though that is also useful, of course!).

more sophisticated file systems, like SGI's XFS, use more complicated tree-based structures [S+96].

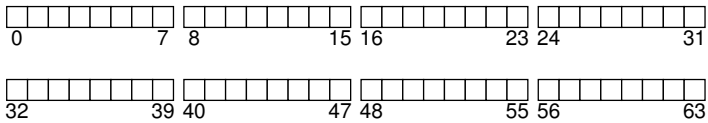
The second aspect of a file system is its **access methods**. How does it map the calls made by a process, such as `open()`, `read()`, `write()`, etc., onto its structures? Which structures are read during the execution of a particular system call? Which are written? How efficiently are all of these steps performed?

If you understand the data structures and access methods of a file system, you have developed a good mental model of how it truly works, a key part of the systems mindset. Try to work on developing your mental model as we delve into our first implementation.

40.2 Overall Organization

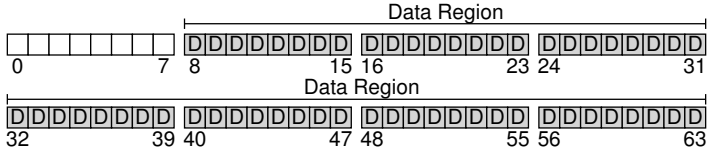
We now develop the overall on-disk organization of the data structures of the vsfs file system. The first thing we'll need to do is divide the disk into **blocks**; simple file systems use just one block size, and that's exactly what we'll do here. Let's choose a commonly-used size of 4 KB.

Thus, our view of the disk partition where we're building our file system is simple: a series of blocks, each of size 4 KB. The blocks are addressed from 0 to $N - 1$, in a partition of size N 4-KB blocks. Assume we have a really small disk, with just 64 blocks:



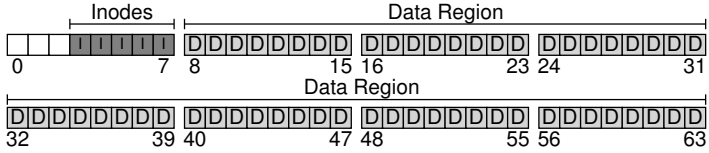
Let's now think about what we need to store in these blocks to build a file system. Of course, the first thing that comes to mind is user data. In fact, most of the space in any file system is (and should be) user data. Let's call the region of the disk we use for user data the **data region**, and,

again for simplicity, reserve a fixed portion of the disk for these blocks, say the last 56 of 64 blocks on the disk:



As we learned about (a little) last chapter, the file system has to track information about each file. This information is a key piece of **metadata**, and tracks things like which data blocks (in the data region) comprise a file, the size of the file, its owner and access rights, access and modify times, and other similar kinds of information. To store this information, file systems usually have a structure called an **inode** (we'll read more about inodes below).

To accommodate inodes, we'll need to reserve some space on the disk for them as well. Let's call this portion of the disk the **inode table**, which simply holds an array of on-disk inodes. Thus, our on-disk image now looks like this picture, assuming that we use 5 of our 64 blocks for inodes (denoted by I's in the diagram):

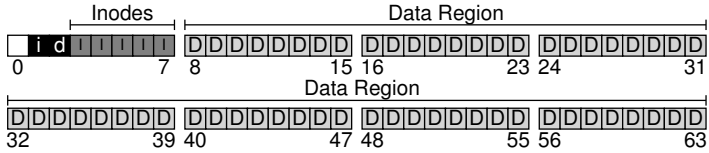


We should note here that inodes are typically not that big, for example 128 or 256 bytes. Assuming 256 bytes per inode, a 4-KB block can hold 16 inodes, and our file system above contains 80 total inodes. In our simple file system, built on a tiny 64-block partition, this number represents the maximum number of files we can have in our file system; however, do note that the same file system, built on a larger disk, could simply allocate a larger inode table and thus accommodate more files.

Our file system thus far has data blocks (D), and inodes (I), but a few things are still missing. One primary component that is still needed, as you might have guessed, is some way to track whether inodes or data blocks are free or allocated. Such **allocation structures** are thus a requisite element in any file system.

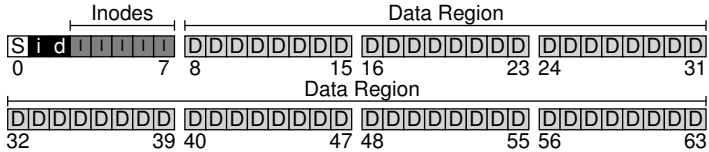
Many allocation-tracking methods are possible, of course. For example, we could use a **free list** that points to the first free block, which then points to the next free block, and so forth. We instead choose a simple and popular structure known as a **bitmap**, one for the data region (the **data bitmap**), and one for the inode table (the **inode bitmap**). A bitmap is a

simple structure: each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1). And thus our new on-disk layout, with an inode bitmap (i) and a data bitmap (d):



You may notice that it is a bit of overkill to use an entire 4-KB block for these bitmaps; such a bitmap can track whether 32K objects are allocated, and yet we only have 80 inodes and 56 data blocks. However, we just use an entire 4-KB block for each of these bitmaps for simplicity.

The careful reader (i.e., the reader who is still awake) may have noticed there is one block left in the design of the on-disk structure of our very simple file system. We reserve this for the **superblock**, denoted by an S in the diagram below. The superblock contains information about this particular file system, including, for example, how many inodes and data blocks are in the file system (80 and 56, respectively in this instance), where the inode table begins (block 3), and so forth. It will likely also include a magic number of some kind to identify the file system type (in this case, vsfs).



Thus, when mounting a file system, the operating system will read the superblock first, to initialize various parameters, and then attach the volume to the file-system tree. When files within the volume are accessed, the system will thus know exactly where to look for the needed on-disk structures.

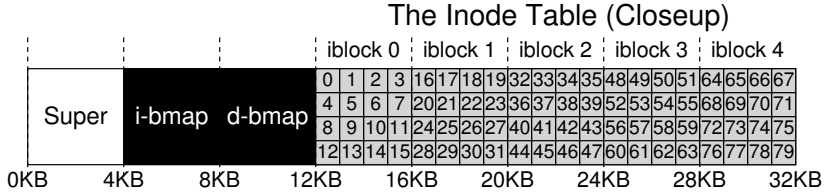
40.3 File Organization: The Inode

One of the most important on-disk structures of a file system is the **inode**; virtually all file systems have a structure similar to this. The name inode is short for **index node**, the historical name given to it in UNIX [RT74] and possibly earlier systems, used because these nodes were originally arranged in an array, and the array *indexed* into when accessing a particular inode.

ASIDE: DATA STRUCTURE — THE INODE

The **inode** is the generic name that is used in many file systems to describe the structure that holds the metadata for a given file, such as its length, permissions, and the location of its constituent blocks. The name goes back at least as far as UNIX (and probably further back to Multics if not earlier systems); it is short for **index node**, as the inode number is used to index into an array of on-disk inodes in order to find the inode of that number. As we'll see, design of the inode is one key part of file system design. Most modern systems have some kind of structure like this for every file they track, but perhaps call them different things (such as dnodes, fnodes, etc.).

Each inode is implicitly referred to by a number (called the **i-number**), which we've earlier called the **low-level name** of the file. In vsfs (and other simple file systems), given an i-number, you should directly be able to calculate where on the disk the corresponding inode is located. For example, take the inode table of vsfs as above: 20KB in size (five 4KB blocks) and thus consisting of 80 inodes (assuming each inode is 256 bytes); further assume that the inode region starts at 12KB (i.e, the superblock starts at 0KB, the inode bitmap is at address 4KB, the data bitmap at 8KB, and thus the inode table comes right after). In vsfs, we thus have the following layout for the beginning of the file system partition (in closeup view):



To read inode number 32, the file system would first calculate the offset into the inode region ($32 \cdot sizeof(inode)$ or 8192), add it to the start address of the inode table on disk (`inodeStartAddr` = $12KB$), and thus arrive upon the correct byte address of the desired block of inodes: $20KB$. Recall that disks are not byte addressable, but rather consist of a large number of addressable sectors, usually 512 bytes. Thus, to fetch the block of inodes that contains inode 32, the file system would issue a read to sector $\frac{20 \times 1024}{512}$, or 40, to fetch the desired inode block. More generally, the sector address `sector` of the inode block can be calculated as follows:

```
blk    = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Inside each inode is virtually all of the information you need about a file: its *type* (e.g., regular file, directory, etc.), its *size*, the number of *blocks* allocated to it, *protection information* (such as who owns the file, as well

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

Figure 40.1: **Simplified Ext2 Inode**

as who can access it), some *time* information, including when the file was created, modified, or last accessed, as well as information about where its data blocks reside on disk (e.g., pointers of some kind). We refer to all such information about a file as **metadata**; in fact, any information inside the file system that isn't pure user data is often referred to as such. An example inode from ext2 [P09] is shown in Figure 40.1[1].

One of the most important decisions in the design of the inode is how it refers to where data blocks are. One simple approach would be to have one or more **direct pointers** (disk addresses) inside the inode; each pointer refers to one disk block that belongs to the file. Such an approach is limited: for example, if you want to have a file that is really big (e.g., bigger than the block size multiplied by the number of direct pointers in the inode), you are out of luck.

The Multi-Level Index

To support bigger files, file system designers have had to introduce different structures within inodes. One common idea is to have a special pointer known as an **indirect pointer**. Instead of pointing to a block that contains user data, it points to a block that contains more pointers, each of which point to user data. Thus, an inode may have some fixed number of direct pointers (e.g., 12), and a single indirect pointer. If a file grows large enough, an indirect block is allocated (from the data-block region of the disk), and the inode's slot for an indirect pointer is set to point to it. Assuming 4-KB blocks and 4-byte disk addresses, that adds another 1024 pointers; the file can grow to be $(12 + 1024) \cdot 4K$ or 4144KB.

[1] Type info is kept in the directory entry, and thus is not found in the inode itself.

TIP: CONSIDER EXTENT-BASED APPROACHES

A different approach is to use **extents** instead of pointers. An extent is simply a disk pointer plus a length (in blocks); thus, instead of requiring a pointer for every block of a file, all one needs is a pointer and a length to specify the on-disk location of a file. Just a single extent is limiting, as one may have trouble finding a contiguous chunk of on-disk free space when allocating a file. Thus, extent-based file systems often allow for more than one extent, thus giving more freedom to the file system during file allocation.

In comparing the two approaches, pointer-based approaches are the most flexible but use a large amount of metadata per file (particularly for large files). Extent-based approaches are less flexible but more compact; in particular, they work well when there is enough free space on the disk and files can be laid out contiguously (which is the goal for virtually any file allocation policy anyhow).

Not surprisingly, in such an approach, you might want to support even larger files. To do so, just add another pointer to the inode: the **double indirect pointer**. This pointer refers to a block that contains pointers to indirect blocks, each of which contain pointers to data blocks. A double indirect block thus adds the possibility to grow files with an additional $1024 \cdot 1024$ or 1-million 4KB blocks, in other words supporting files that are over 4GB in size. You may want even more, though, and we bet you know where this is headed: the **triple indirect pointer**.

Overall, this imbalanced tree is referred to as the **multi-level index** approach to pointing to file blocks. Let's examine an example with twelve direct pointers, as well as both a single and a double indirect block. Assuming a block size of 4 KB, and 4-byte pointers, this structure can accommodate a file of just over 4 GB in size (i.e., $(12 + 1024 + 1024^2) \times 4\ KB$). Can you figure out how big of a file can be handled with the addition of a triple-indirect block? (hint: pretty big)

Many file systems use a multi-level index, including commonly-used file systems such as Linux ext2 [P09] and ext3, NetApp's WAFL, as well as the original UNIX file system. Other file systems, including SGI XFS and Linux ext4, use **extents** instead of simple pointers; see the earlier aside for details on how extent-based schemes work (they are akin to segments in the discussion of virtual memory).

You might be wondering: why use an imbalanced tree like this? Why not a different approach? Well, as it turns out, many researchers have studied file systems and how they are used, and virtually every time they find certain "truths" that hold across the decades. One such finding is that *most files are small*. This imbalanced design reflects such a reality; if most files are indeed small, it makes sense to optimize for this case. Thus, with a small number of direct pointers (12 is a typical number), an inode

Most files are small	~2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of space
File systems contain lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Figure 40.2: **File System Measurement Summary**

can directly point to 48 KB of data, needing one (or more) indirect blocks for larger files. See Agrawal et. al [A+07] for a recent-ish study; Figure 40.2 summarizes those results.

Of course, in the space of inode design, many other possibilities exist; after all, the inode is just a data structure, and any data structure that stores the relevant information, and can query it effectively, is sufficient. As file system software is readily changed, you should be willing to explore different designs should workloads or technologies change.

40.4 Directory Organization

In vsfs (as in many file systems), directories have a simple organization; a directory basically just contains a list of (entry name, inode number) pairs. For each file or directory in a given directory, there is a string and a number in the data block(s) of the directory. For each string, there may also be a length (assuming variable-sized names).

For example, assume a directory `dir` (inode number 5) has three files in it (`foo`, `bar`, and `foobar_is_a_pretty_longname`), with inode numbers 12, 13, and 24 respectively. The on-disk data for `dir` might look like:

```
inum | reclen | strlen | name
   5      12        2     .
   2      12        3     ..
  12      12        4     foo
  13      12        4     bar
  24      36       28     foobar_is_a_pretty_longname
```

In this example, each entry has an inode number, record length (the total bytes for the name plus any left over space), string length (the actual length of the name), and finally the name of the entry. Note that each directory has two extra entries, `.` "dot" and `..` "dot-dot"; the dot directory is just the current directory (in this example, `dir`), whereas dot-dot is the parent directory (in this case, the root).

Deleting a file (e.g., calling `unlink()`) can leave an empty space in the middle of the directory, and hence there should be some way to mark that as well (e.g., with a reserved inode number such as zero). Such a delete is one reason the record length is used: a new entry may reuse an old, bigger entry and thus have extra space within.

ASIDE: LINKED-BASED APPROACHES

Another simpler approach in designing inodes is to use a **linked list**. Thus, inside an inode, instead of having multiple pointers, you just need one, to point to the first block of the file. To handle larger files, add another pointer at the end of that data block, and so on, and thus you can support large files.

As you might have guessed, linked file allocation performs poorly for some workloads; think about reading the last block of a file, for example, or just doing random access. Thus, to make linked allocation work better, some systems will keep an in-memory table of link information, instead of storing the next pointers with the data blocks themselves. The table is indexed by the address of a data block D; the content of an entry is simply D's next pointer, i.e., the address of the next block in a file which follows D. A null-value could be there too (indicating an end-of-file), or some other marker to indicate that a particular block is free. Having such a table of next pointers makes it so that a linked allocation scheme can effectively do random file accesses, simply by first scanning through the (in memory) table to find the desired block, and then accessing (on disk) it directly.

Does such a table sound familiar? What we have described is the basic structure of what is known as the **file allocation table**, or **FAT** file system. Yes, this classic old Windows file system, before NTFS [C94], is based on a simple linked-based allocation scheme. There are other differences from a standard UNIX file system too; for example, there are no inodes per se, but rather directory entries which store metadata about a file and refer directly to the first block of said file, which makes creating hard links impossible. See Brouwer [B02] for more of the inelegant details.

You might be wondering where exactly directories are stored. Often, file systems treat directories as a special type of file. Thus, a directory has an inode, somewhere in the inode table (with the type field of the inode marked as "directory" instead of "regular file"). The directory has data blocks pointed to by the inode (and perhaps, indirect blocks); these data blocks live in the data block region of our simple file system. Our on-disk structure thus remains unchanged.

We should also note again that this simple linear list of directory entries is not the only way to store such information. As before, any data structure is possible. For example, XFS [S+96] stores directories in B-tree form, making file create operations (which have to ensure that a file name has not been used before creating it) faster than systems with simple lists that must be scanned in their entirety.

ASIDE: FREE SPACE MANAGEMENT

There are many ways to manage free space; bitmaps are just one way. Some early file systems used **free lists**, where a single pointer in the super block was kept to point to the first free block; inside that block the next free pointer was kept, thus forming a list through the free blocks of the system. When a block was needed, the head block was used and the list updated accordingly.

Modern file systems use more sophisticated data structures. For example, SGI's XFS [S+96] uses some form of a **B-tree** to compactly represent which chunks of the disk are free. As with any data structure, different time-space trade-offs are possible.

40.5 Free Space Management

A file system must track which inodes and data blocks are free, and which are not, so that when a new file or directory is allocated, it can find space for it. Thus **free space management** is important for all file systems. In vsfs, we have two simple bitmaps for this task.

For example, when we create a file, we will have to allocate an inode for that file. The file system will thus search through the bitmap for an inode that is free, and allocate it to the file; the file system will have to mark the inode as used (with a 1) and eventually update the on-disk bitmap with the correct information. A similar set of activities take place when a data block is allocated.

Some other considerations might also come into play when allocating data blocks for a new file. For example, some Linux file systems, such as ext2 and ext3, will look for a sequence of blocks (say 8) that are free when a new file is created and needs data blocks; by finding such a sequence of free blocks, and then allocating them to the newly-created file, the file system guarantees that a portion of the file will be contiguous on the disk, thus improving performance. Such a **pre-allocation** policy is thus a commonly-used heuristic when allocating space for data blocks.

40.6 Access Paths: Reading and Writing

Now that we have some idea of how files and directories are stored on disk, we should be able to follow the flow of operation during the activity of reading or writing a file. Understanding what happens on this **access path** is thus the second key in developing an understanding of how a file system works; pay attention!

For the following examples, let us assume that the file system has been mounted and thus that the superblock is already in memory. Everything else (i.e., inodes, directories) is still on the disk.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**			**read**				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

Figure 40.3: **File Read Timeline (Time Increasing Downward)**

Reading A File From Disk

In this simple example, let us first assume that you want to simply open a file (e.g., `/foo/bar`), read it, and then close it. For this simple example, let's assume the file is just 12KB in size (i.e., 3 blocks).

When you issue an `open("/foo/bar", O_RDONLY)` call, the file system first needs to find the inode for the file `bar`, to obtain some basic information about the file (permissions information, file size, etc.). To do so, the file system must be able to find the inode, but all it has right now is the full pathname. The file system must **traverse** the pathname and thus locate the desired inode.

All traversals begin at the root of the file system, in the **root directory** which is simply called `/`. Thus, the first thing the FS will read from disk is the inode of the root directory. But where is this inode? To find an inode, we must know its i-number. Usually, we find the i-number of a file or directory in its parent directory; the root has no parent (by definition). Thus, the root inode number must be "well known"; the FS must know what it is when the file system is mounted. In most UNIX file systems, the root inode number is 2. Thus, to begin the process, the FS reads in the block that contains inode number 2 (the first inode block).

Once the inode is read in, the FS can look inside of it to find pointers to data blocks, which contain the contents of the root directory. The FS will thus use these on-disk pointers to read through the directory, in this case looking for an entry for `foo`. By reading in one or more directory data blocks, it will find the entry for foo; once found, the FS will also have found the inode number of `foo` (say it is 44) which it will need next.

The next step is to recursively traverse the pathname until the desired inode is found. In this example, the FS reads the block containing the

ASIDE: READS DON'T ACCESS ALLOCATION STRUCTURES
We've seen many students get confused by allocation structures such as bitmaps. In particular, many often think that when you are simply reading a file, and not allocating any new blocks, that the bitmap will still be consulted. This is not true! Allocation structures, such as bitmaps, are only accessed when allocation is needed. The inodes, directories, and indirect blocks have all the information they need to complete a read request; there is no need to make sure a block is allocated when the inode already points to it.

inode of `foo` and then its directory data, finally finding the inode number of `bar`. The final step of `open()` is to read `bar`'s inode into memory; the FS then does a final permissions check, allocates a file descriptor for this process in the per-process open-file table, and returns it to the user.

Once open, the program can then issue a `read()` system call to read from the file. The first read (at offset 0 unless `lseek()` has been called) will thus read in the first block of the file, consulting the inode to find the location of such a block; it may also update the inode with a new last-accessed time. The read will further update the in-memory open file table for this file descriptor, updating the file offset such that the next read will read the second file block, etc.

At some point, the file will be closed. There is much less work to be done here; clearly, the file descriptor should be deallocated, but for now, that is all the FS really needs to do. No disk I/Os take place.

A depiction of this entire process is found in Figure 40.3 (page 11); time increases downward in the figure. In the figure, the open causes numerous reads to take place in order to finally locate the inode of the file. Afterwards, reading each block requires the file system to first consult the inode, then read the block, and then update the inode's last-accessed-time field with a write. Spend some time and understand what is going on.

Also note that the amount of I/O generated by the open is proportional to the length of the pathname. For each additional directory in the path, we have to read its inode as well as its data. Making this worse would be the presence of large directories; here, we only have to read one block to get the contents of a directory, whereas with a large directory, we might have to read many data blocks to find the desired entry. Yes, life can get pretty bad when reading a file; as you're about to find out, writing out a file (and especially, creating a new one) is even worse.

Writing A File To Disk

Writing to a file is a similar process. First, the file must be opened (as above). Then, the application can issue `write()` calls to update the file with new contents. Finally, the file is closed.

Unlike reading, writing to the file may also **allocate** a block (unless the block is being overwritten, for example). When writing out a new file, each write not only has to write data to disk but has to first decide

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
create (/foo/bar)		**read** **write**	**read**	**read**	**read** **write** **write**	**read**	**read** **write**			
write()	**read** **write**				**read** **write**			**write**		
write()	**read** **write**				**read** **write**				**write**	
write()	**read** **write**				**read** **write**					**write**

Figure 40.4: **File Creation Timeline (Time Increasing Downward)**

which block to allocate to the file and thus update other structures of the disk accordingly (e.g., the data bitmap and inode). Thus, each write to a file logically generates five I/Os: one to read the data bitmap (which is then updated to mark the newly-allocated block as used), one to write the bitmap (to reflect its new state to disk), two more to read and then write the inode (which is updated with the new block's location), and finally one to write the actual block itself.

The amount of write traffic is even worse when one considers a simple and common operation such as file creation. To create a file, the file system must not only allocate an inode, but also allocate space within the directory containing the new file. The total amount of I/O traffic to do so is quite high: one read to the inode bitmap (to find a free inode), one write to the inode bitmap (to mark it allocated), one write to the new inode itself (to initialize it), one to the data of the directory (to link the high-level name of the file to its inode number), and one read and write to the directory inode to update it. If the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too. All that just to create a file!

Let's look at a specific example, where the file `/foo/bar` is created, and three blocks are written to it. Figure 40.4 (page 13) shows what happens during the `open()` (which creates the file) and during each of three 4KB writes.

In the figure, reads and writes to the disk are grouped under which system call caused them to occur, and the rough ordering they might take place in goes from top to bottom of the figure. You can see how much work it is to create the file: 10 I/Os in this case, to walk the pathname and then finally create the file. You can also see that each allocating write costs 5 I/Os: a pair to read and update the inode, another pair to read and update the data bitmap, and then finally the write of the data itself. How can a file system accomplish any of this with reasonable efficiency?

THE CRUX: HOW TO REDUCE FILE SYSTEM I/O COSTS

Even the simplest of operations like opening, reading, or writing a file incurs a huge number of I/O operations, scattered over the disk. What can a file system do to reduce the high costs of doing so many I/Os?

40.7 Caching and Buffering

As the examples above show, reading and writing files can be expensive, incurring many I/Os to the (slow) disk. To remedy what would clearly be a huge performance problem, most file systems aggressively use system memory (DRAM) to cache important blocks.

Imagine the open example above: without caching, every file open would require at least two reads for every level in the directory hierarchy (one to read the inode of the directory in question, and at least one to read its data). With a long pathname (e.g., `/1/2/3/ ... /100/file.txt`), the file system would literally perform hundreds of reads just to open the file!

Early file systems thus introduced a **fixed-size cache** to hold popular blocks. As in our discussion of virtual memory, strategies such as **LRU** and different variants would decide which blocks to keep in cache. This fixed-size cache would usually be allocated at boot time to be roughly 10% of total memory.

This **static partitioning** of memory, however, can be wasteful; what if the file system doesn't need 10% of memory at a given point in time? With the fixed-size approach described above, unused pages in the file cache cannot be re-purposed for some other use, and thus go to waste.

Modern systems, in contrast, employ a **dynamic partitioning** approach. Specifically, many modern operating systems integrate virtual memory pages and file system pages into a **unified page cache** [S00]. In this way, memory can be allocated more flexibly across virtual memory and file system, depending on which needs more memory at a given time.

Now imagine the file open example with caching. The first open may generate a lot of I/O traffic to read in directory inode and data, but sub-

TIP: UNDERSTAND STATIC VS. DYNAMIC PARTITIONING

When dividing a resource among different clients/users, you can use either **static partitioning** or **dynamic partitioning**. The static approach simply divides the resource into fixed proportions once; for example, if there are two possible users of memory, you can give some fixed fraction of memory to one user, and the rest to the other. The dynamic approach is more flexible, giving out differing amounts of the resource over time; for example, one user may get a higher percentage of disk bandwidth for a period of time, but then later, the system may switch and decide to give a different user a larger fraction of available disk bandwidth.

Each approach has its advantages. Static partitioning ensures each user receives some share of the resource, usually delivers more predictable performance, and is often easier to implement. Dynamic partitioning can achieve better utilization (by letting resource-hungry users consume otherwise idle resources), but can be more complex to implement, and can lead to worse performance for users whose idle resources get consumed by others and then take a long time to reclaim when needed. As is often the case, there is no best method; rather, you should think about the problem at hand and decide which approach is most suitable. Indeed, shouldn't you always be doing that?

sequent file opens of that same file (or files in the same directory) will mostly hit in the cache and thus no I/O is needed.

Let us also consider the effect of caching on writes. Whereas read I/O can be avoided altogether with a sufficiently large cache, write traffic has to go to disk in order to become persistent. Thus, a cache does not serve as the same kind of filter on write traffic that it does for reads. That said, **write buffering** (as it is sometimes called) certainly has a number of performance benefits. First, by delaying writes, the file system can **batch** some updates into a smaller set of I/Os; for example, if an inode bitmap is updated when one file is created and then updated moments later as another file is created, the file system saves an I/O by delaying the write after the first update. Second, by buffering a number of writes in memory, the system can then **schedule** the subsequent I/Os and thus increase performance. Finally, some writes are avoided altogether by delaying them; for example, if an application creates a file and then deletes it, delaying the writes to reflect the file creation to disk **avoids** them entirely. In this case, laziness (in writing blocks to disk) is a virtue.

For the reasons above, most modern file systems buffer writes in memory for anywhere between five and thirty seconds, representing yet another trade-off: if the system crashes before the updates have been propagated to disk, the updates are lost; however, by keeping writes in memory longer, performance can be improved by batching, scheduling, and even avoiding writes.

TIP: UNDERSTAND THE DURABILITY/PERFORMANCE TRADE-OFF
Storage systems often present a durability/performance trade-off to users. If the user wishes data that is written to be immediately durable, the system must go through the full effort of committing the newly-written data to disk, and thus the write is slow (but safe). However, if the user can tolerate the loss of a little data, the system can buffer writes in memory for some time and write them later to the disk (in the background). Doing so makes writes appear to complete quickly, thus improving perceived performance; however, if a crash occurs, writes not yet committed to disk will be lost, and hence the trade-off. To understand how to make this trade-off properly, it is best to understand what the application using the storage system requires; for example, while it may be tolerable to lose the last few images downloaded by your web browser, losing part of a database transaction that is adding money to your bank account may be less tolerable. Unless you're rich, of course; in that case, why do you care so much about hoarding every last penny?

Some applications (such as databases) don't enjoy this trade-off. Thus, to avoid unexpected data loss due to write buffering, they simply force writes to disk, by calling `fsync()`, by using **direct I/O** interfaces that work around the cache, or by using the **raw disk** interface and avoiding the file system altogether[2]. While most applications live with the trade-offs made by the file system, there are enough controls in place to get the system to do what you want it to, should the default not be satisfying.

40.8 Summary

We have seen the basic machinery required in building a file system. There needs to be some information about each file (metadata), usually stored in a structure called an inode. Directories are just a specific type of file that store name→inode-number mappings. And other structures are needed too; for example, file systems often use a structure such as a bitmap to track which inodes or data blocks are free or allocated.

The terrific aspect of file system design is its freedom; the file systems we explore in the coming chapters each take advantage of this freedom to optimize some aspect of the file system. There are also clearly many policy decisions we have left unexplored. For example, when a new file is created, where should it be placed on disk? This policy and others will also be the subject of future chapters. Or will they?[3]

[2] Take a database class to learn more about old-school databases and their former insistence on avoiding the OS and controlling everything themselves. But watch out! Those database types are always trying to bad mouth the OS. Shame on you, database people. Shame.

[3] Cue mysterious music that gets you even more intrigued about the topic of file systems.

References

[A+07] "A Five-Year Study of File-System Metadata" by Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch. FAST '07, San Jose, California, February 2007. *An excellent recent analysis of how file systems are actually used. Use the bibliography within to follow the trail of file-system analysis papers back to the early 1980s.*

[B07] "ZFS: The Last Word in File Systems" by Jeff Bonwick and Bill Moore. Available from: `http://www.ostep.org/Citations/zfs_last.pdf`. *One of the most recent important file systems, full of features and awesomeness. We should have a chapter on it, and perhaps soon will.*

[B02] "The FAT File System" by Andries Brouwer. September, 2002. Available online at: `http://www.win.tue.nl/%7eaeb/linux/fs/fat/fat.html`. *A nice clean description of FAT. The file system kind, not the bacon kind. Though you have to admit, bacon fat probably tastes better.*

[C94] "Inside the Windows NT File System" by Helen Custer. Microsoft Press, 1994. *A short book about NTFS; there are probably ones with more technical details elsewhere.*

[H+88] "Scale and Performance in a Distributed File System" by John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West.. ACM TOCS, Volume 6:1, February 1988. *A classic distributed file system; we'll be learning more about it later, don't worry.*

[P09] "The Second Extended File System: Internal Layout" by Dave Poirier. 2009. Available: `http://www.nongnu.org/ext2-doc/ext2.html`. *Some details on ext2, a very simple Linux file system based on FFS, the Berkeley Fast File System. We'll be reading about it in the next chapter.*

[RT74] "The UNIX Time-Sharing System" by M. Ritchie, K. Thompson. CACM Volume 17:7, 1974. *The original paper about UNIX. Read it to see the underpinnings of much of modern operating systems.*

[S00] "UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD" by Chuck Silvers. FREENIX, 2000. *A nice paper about NetBSD's integration of file-system buffer caching and the virtual-memory page cache. Many other systems do the same type of thing.*

[S+96] "Scalability in the XFS File System" by Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, Geoff Peck. USENIX '96, January 1996, San Diego, California. *The first attempt to make scalability of operations, including things like having millions of files in a directory, a central focus. A great example of pushing an idea to the extreme. The key idea behind this file system: everything is a tree. We should have a chapter on this file system too.*

Homework (Simulation)

Use this tool, `vsfs.py`, to study how file system state changes as various operations take place. The file system begins in an empty state, with just a root directory. As the simulation takes place, various operations are performed, thus slowly changing the on-disk state of the file system. See the README for details.

Questions

1. Run the simulator with some different random seeds (say 17, 18, 19, 20), and see if you can figure out which operations must have taken place between each state change.
2. Now do the same, using different random seeds (say 21, 22, 23, 24), except run with the `-r` flag, thus making you guess the state change while being shown the operation. What can you conclude about the inode and data-block allocation algorithms, in terms of which blocks they prefer to allocate?
3. Now reduce the number of data blocks in the file system, to very low numbers (say two), and run the simulator for a hundred or so requests. What types of files end up in the file system in this highly-constrained layout? What types of operations would fail?
4. Now do the same, but with inodes. With very few inodes, what types of operations can succeed? Which will usually fail? What is the final state of the file system likely to be?

41

Locality and The Fast File System

When the UNIX operating system was first introduced, the UNIX wizard himself Ken Thompson wrote the first file system. Let's call that the "old UNIX file system", and it was really simple. Basically, its data structures looked like this on the disk:

The super block (S) contained information about the entire file system: how big the volume is, how many inodes there are, a pointer to the head of a free list of blocks, and so forth. The inode region of the disk contained all the inodes for the file system. Finally, most of the disk was taken up by data blocks.

The good thing about the old file system was that it was simple, and supported the basic abstractions the file system was trying to deliver: files and the directory hierarchy. This easy-to-use system was a real step forward from the clumsy, record-based storage systems of the past, and the directory hierarchy was a true advance over simpler, one-level hierarchies provided by earlier systems.

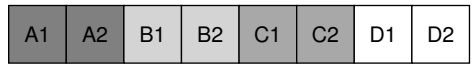
41.1 The Problem: Poor Performance

The problem: performance was terrible. As measured by Kirk McKusick and his colleagues at Berkeley [MJLF84], performance started off bad and got worse over time, to the point where the file system was delivering only 2% of overall disk bandwidth!

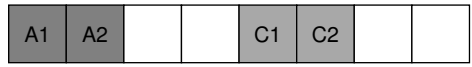
The main issue was that the old UNIX file system treated the disk like it was a random-access memory; data was spread all over the place without regard to the fact that the medium holding the data was a disk, and thus had real and expensive positioning costs. For example, the data blocks of a file were often very far away from its inode, thus inducing an expensive seek whenever one first read the inode and then the data blocks of a file (a pretty common operation).

Worse, the file system would end up getting quite **fragmented**, as the free space was not carefully managed. The free list would end up pointing to a bunch of blocks spread across the disk, and as files got allocated, they would simply take the next free block. The result was that a logically contiguous file would be accessed by going back and forth across the disk, thus reducing performance dramatically.

For example, imagine the following data block region, which contains four files (A, B, C, and D), each of size 2 blocks:



If B and D are deleted, the resulting layout is:



As you can see, the free space is fragmented into two chunks of two blocks, instead of one nice contiguous chunk of four. Let's say you now wish to allocate a file E, of size four blocks:

You can see what happens: E gets spread across the disk, and as a result, when accessing E, you don't get peak (sequential) performance from the disk. Rather, you first read E1 and E2, then seek, then read E3 and E4. This fragmentation problem happened all the time in the old UNIX file system, and it hurt performance. A side note: this problem is exactly what disk **defragmentation** tools help with; they reorganize on-disk data to place files contiguously and make free space for one or a few contiguous regions, moving data around and then rewriting inodes and such to reflect the changes.

One other problem: the original block size was too small (512 bytes). Thus, transferring data from the disk was inherently inefficient. Smaller blocks were good because they minimized **internal fragmentation** (waste within the block), but bad for transfer as each block might require a positioning overhead to reach it. Thus, the problem:

THE CRUX:
HOW TO ORGANIZE ON-DISK DATA TO IMPROVE PERFORMANCE

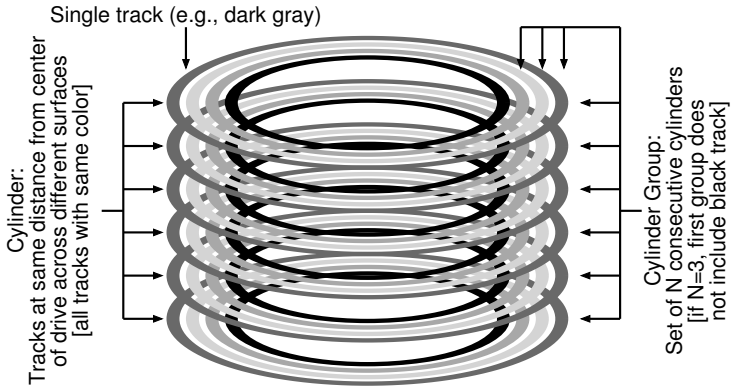
How can we organize file system data structures so as to improve performance? What types of allocation policies do we need on top of those data structures? How do we make the file system "disk aware"?

41.2 FFS: Disk Awareness Is The Solution

A group at Berkeley decided to build a better, faster file system, which they cleverly called the **Fast File System (FFS)**. The idea was to design the file system structures and allocation policies to be "disk aware" and thus improve performance, which is exactly what they did. FFS thus ushered in a new era of file system research; by keeping the same *interface* to the file system (the same APIs, including `open()`, `read()`, `write()`, `close()`, and other file system calls) but changing the internal *implementation*, the authors paved the path for new file system construction, work that continues today. Virtually all modern file systems adhere to the existing interface (and thus preserve compatibility with applications) while changing their internals for performance, reliability, or other reasons.

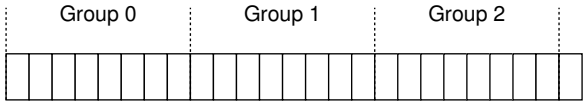
41.3 Organizing Structure: The Cylinder Group

The first step was to change the on-disk structures. FFS divides the disk into a number of **cylinder groups**. A single **cylinder** is a set of tracks on different surfaces of a hard drive that are the same distance from the center of the drive; it is called a cylinder because of its clear resemblance to the so-called geometrical shape. FFS aggregates N consecutive cylinders into a group, and thus the entire disk can thus be viewed as a collection of cylinder groups. Here is a simple example, showing the four outer most tracks of a drive with six platters, and a cylinder group that consists of three cylinders:



Note that modern drives do not export enough information for the file system to truly understand whether a particular cylinder is in use; as discussed previously [AD14a], disks export a logical address space of blocks and hide details of their geometry from clients. Thus, modern file

systems (such as Linux ext2, ext3, and ext4) instead organize the drive into **block groups**, each of which is just a consecutive portion of the disk's address space. The picture below illustrates an example where every 8 blocks are organized into a different block group (note that real groups would consist of many more blocks):



Whether you call them cylinder groups or block groups, these groups are the central mechanism that FFS uses to improve performance. Critically, by placing two files within the same group, FFS can ensure that accessing one after the other will not result in long seeks across the disk.

To use these groups to store files and directories, FFS needs to have the ability to place files and directories into a group, and track all necessary information about them therein. To do so, FFS includes all the structures you might expect a file system to have within each group, e.g., space for inodes, data blocks, and some structures to track whether each of those are allocated or free. Here is a depiction of what FFS keeps within a single cylinder group:

Let's now examine the components of this single cylinder group in more detail. FFS keeps a copy of the **super block** (S) in each group for reliability reasons. The super block is needed to mount the file system; by keeping multiple copies, if one copy becomes corrupt, you can still mount and access the file system by using a working replica.

Within each group, FFS needs to track whether the inodes and data blocks of the group are allocated. A per-group **inode bitmap** (ib) and **data bitmap** (db) serve this role for inodes and data blocks in each group. Bitmaps are an excellent way to manage free space in a file system because it is easy to find a large chunk of free space and allocate it to a file, perhaps avoiding some of the fragmentation problems of the free list in the old file system.

Finally, the **inode** and **data block** regions are just like those in the previous very-simple file system (VSFS). Most of each cylinder group, as usual, is comprised of data blocks.

ASIDE: FFS FILE CREATION

As an example, think about what data structures must be updated when a file is created; assume, for this example, that the user creates a new file `/foo/bar.txt` and that the file is one block long (4KB). The file is new, and thus needs a new inode; thus, both the inode bitmap and the newly-allocated inode will be written to disk. The file also has data in it and thus it too must be allocated; the data bitmap and a data block will thus (eventually) be written to disk. Hence, at least four writes to the current cylinder group will take place (recall that these writes may be buffered in memory for a while before they take place). But this is not all! In particular, when creating a new file, you must also place the file in the file-system hierarchy, i.e., the directory must be updated. Specifically, the parent directory `foo` must be updated to add the entry for `bar.txt`; this update may fit in an existing data block of `foo` or require a new block to be allocated (with associated data bitmap). The inode of `foo` must also be updated, both to reflect the new length of the directory as well as to update time fields (such as last-modified-time). Overall, it is a lot of work just to create a new file! Perhaps next time you do so, you should be more thankful, or at least surprised that it all works so well.

41.4 Policies: How To Allocate Files and Directories

With this group structure in place, FFS now has to decide how to place files and directories and associated metadata on disk to improve performance. The basic mantra is simple: *keep related stuff together* (and its corollary, *keep unrelated stuff far apart*).

Thus, to obey the mantra, FFS has to decide what is "related" and place it within the same block group; conversely, unrelated items should be placed into different block groups. To achieve this end, FFS makes use of a few simple placement heuristics.

The first is the placement of directories. FFS employs a simple approach: find the cylinder group with a low number of allocated directories (to balance directories across groups) and a high number of free inodes (to subsequently be able to allocate a bunch of files), and put the directory data and inode in that group. Of course, other heuristics could be used here (e.g., taking into account the number of free data blocks).

For files, FFS does two things. First, it makes sure (in the general case) to allocate the data blocks of a file in the same group as its inode, thus preventing long seeks between inode and data (as in the old file system). Second, it places all files that are in the same directory in the cylinder group of the directory they are in. Thus, if a user creates four files, `/a/b`, `/a/c`, `/a/d`, and `b/f`, FFS would try to place the first three near one another (same group) and the fourth far away (in some other group).

Let's look at an example of such an allocation. In the example, assume that there are only 10 inodes and 10 data blocks in each group (both

unrealistically small numbers), and that the three directories (the root directory `/`, `/a`, and `/b`) and four files (`/a/c`, `/a/d`, `/a/e`, `/b/f`) are placed within them per the FFS policies. Assume the regular files are each two blocks in size, and that the directories have just a single block of data. For this figure, we use the obvious symbols for each file or directory (i.e., `/` for the root directory, `a` for `/a`, `f` for `/b/f`, and so forth).

```
group inodes     data
    0 /--------- /---------
    1 acde------ accddee---
    2 bf-------- bff-------
    3 ---------- ----------
    4 ---------- ----------
    5 ---------- ----------
    6 ---------- ----------
    7 ---------- ----------
```

Note that the FFS policy does two positive things: the data blocks of each file are near each file's inode, and files in the same directory are near one another (namely, `/a/c`, `/a/d`, and `/a/e` are all in Group 1, and directory `/b` and its file `/b/f` are near one another in Group 2).

In contrast, let's now look at an inode allocation policy that simply spreads inodes across groups, trying to ensure that no group's inode table fills up quickly. The final allocation might thus look something like this:

```
group inodes     data
    0 /--------- /---------
    1 a--------- a---------
    2 b--------- b---------
    3 c--------- cc--------
    4 d--------- dd--------
    5 e--------- ee--------
    6 f--------- ff--------
    7 ---------- ----------
```

As you can see from the figure, while this policy does indeed keep file (and directory) data near its respective inode, files within a directory are arbitrarily spread around the disk, and thus name-based locality is not preserved. Access to files `/a/c`, `/a/d`, and `/a/e` now spans three groups instead of one as per the FFS approach.

The FFS policy heuristics are not based on extensive studies of file-system traffic or anything particularly nuanced; rather, they are based on good old-fashioned **common sense** (isn't that what CS stands for after all?)[1]. Files in a directory *are* often accessed together: imagine compiling a bunch of files and then linking them into a single executable. Be-

[1]Some people refer to common sense as **horse sense**, especially people who work regularly with horses. However, we have a feeling that this idiom may be lost as the "mechanized horse", a.k.a. the car, gains in popularity. What will they invent next? A flying machine??!!

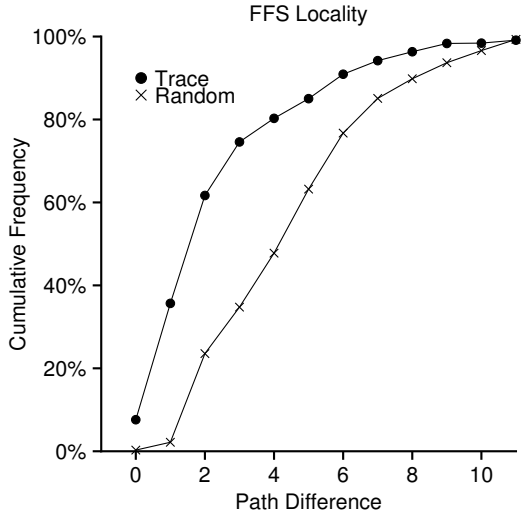


Figure 41.1: **FFS Locality For SEER Traces**

cause such namespace-based locality exists, FFS will often improve performance, making sure that seeks between related files are nice and short.

41.5 Measuring File Locality

To understand better whether these heuristics make sense, let's analyze some traces of file system access and see if indeed there is namespace locality. For some reason, there doesn't seem to be a good study of this topic in the literature.

Specifically, we'll use the SEER traces [K94] and analyze how "far away" file accesses were from one another in the directory tree. For example, if file `f` is opened, and then re-opened next in the trace (before any other files are opened), the distance between these two opens in the directory tree is zero (as they are the same file). If a file `f` in directory `dir` (i.e., `dir/f`) is opened, and followed by an open of file `g` in the same directory (i.e., `dir/g`), the distance between the two file accesses is one, as they share the same directory but are not the same file. Our distance metric, in other words, measures how far up the directory tree you have to travel to find the *common ancestor* of two files; the closer they are in the tree, the lower the metric.

Figure 41.1 shows the locality observed in the SEER traces over all workstations in the SEER cluster over the entirety of all traces. The graph plots the difference metric along the x-axis, and shows the cumulative percentage of file opens that were of that difference along the y-axis. Specifically, for the SEER traces (marked "Trace" in the graph), you can see that about 7% of file accesses were to the file that was opened previ-

ously, and that nearly 40% of file accesses were to either the same file or to one in the same directory (i.e., a difference of zero or one). Thus, the FFS locality assumption seems to make sense (at least for these traces).

Interestingly, another 25% or so of file accesses were to files that had a distance of two. This type of locality occurs when the user has structured a set of related directories in a multi-level fashion and consistently jumps between them. For example, if a user has a `src` directory and builds object files (`.o` files) into an `obj` directory, and both of these directories are sub-directories of a main `proj` directory, a common access pattern will be `proj/src/foo.c` followed by `proj/obj/foo.o`. The distance between these two accesses is two, as `proj` is the common ancestor. FFS does *not* capture this type of locality in its policies, and thus more seeking will occur between such accesses.

For comparison, the graph also shows locality for a "Random" trace. The random trace was generated by selecting files from within an existing SEER trace in random order, and calculating the distance metric between these randomly-ordered accesses. As you can see, there is less namespace locality in the random traces, as expected. However, because eventually every file shares a common ancestor (e.g., the root), there is some locality, and thus random is useful as a comparison point.

41.6 The Large-File Exception

In FFS, there is one important exception to the general policy of file placement, and it arises for large files. Without a different rule, a large file would entirely fill the block group it is first placed within (and maybe others). Filling a block group in this manner is undesirable, as it prevents subsequent "related" files from being placed within this block group, and thus may hurt file-access locality.

Thus, for large files, FFS does the following. After some number of blocks are allocated into the first block group (e.g., 12 blocks, or the number of direct pointers available within an inode), FFS places the next "large" chunk of the file (e.g., those pointed to by the first indirect block) in another block group (perhaps chosen for its low utilization). Then, the next chunk of the file is placed in yet another different block group, and so on.

Let's look at some diagrams to understand this policy better. Without the large-file exception, a single large file would place all of its blocks into one part of the disk. We investigate a small example of a file (`/a`) with 30 blocks in an FFS configured with 10 inodes and 40 data blocks per group. Here is the depiction of FFS without the large-file exception:

```
group inodes     data
  0   /a-------- /aaaaaaaaa aaaaaaaaa aaaaaaaaa a---------
  1   ---------- ---------- ---------- ---------- ----------
  2   ---------- ---------- ---------- ---------- ----------
```

As you can see in the picture, `/a` fills up most of the data blocks in Group 0, whereas other groups remain empty. If some other files are now created in the root directory (`/`), there is not much room for their data in the group.

With the large-file exception (here set to five blocks in each chunk), FFS instead spreads the file spread across groups, and the resulting utilization within any one group is not too high:

```
group inodes     data
  0   /a-------- /aaaaa----- ---------- ---------- ----------
  1   ---------- aaaaa----- ---------- ---------- ----------
  2   ---------- aaaaa----- ---------- ---------- ----------
  3   ---------- aaaaa----- ---------- ---------- ----------
  4   ---------- aaaaa----- ---------- ---------- ----------
  5   ---------- aaaaa----- ---------- ---------- ----------
  6   ---------- ---------- ---------- ---------- ----------
```

The astute reader (that's you) will note that spreading blocks of a file across the disk will hurt performance, particularly in the relatively common case of sequential file access (e.g., when a user or application reads chunks 0 through 29 in order). And you are right, oh astute reader of ours! But you can address this problem by choosing chunk size carefully.

Specifically, if the chunk size is large enough, the file system will spend most of its time transferring data from disk and just a (relatively) little time seeking between chunks of the block. This process of reducing an overhead by doing more work per overhead paid is called **amortization** and is a common technique in computer systems.

Let's do an example: assume that the average positioning time (i.e., seek and rotation) for a disk is 10 ms. Assume further that the disk transfers data at 40 MB/s. If your goal was to spend half our time seeking between chunks and half our time transferring data (and thus achieve 50% of peak disk performance), you would thus need to spend 10 ms transferring data for every 10 ms positioning. So the question becomes: how big does a chunk have to be in order to spend 10 ms in transfer? Easy, just use our old friend, math, in particular the dimensional analysis mentioned in the chapter on disks [AD14a]:

$$\frac{40\ \cancel{MB}}{\cancel{sec}} \cdot \frac{1024\ KB}{1\ \cancel{MB}} \cdot \frac{1\ \cancel{sec}}{1000\ \cancel{ms}} \cdot 10\ \cancel{ms} = 409.6\ KB \tag{41.1}$$

Basically, what this equation says is this: if you transfer data at 40 MB/s, you need to transfer only 409.6KB every time you seek in order to spend half your time seeking and half your time transferring. Similarly, you can compute the size of the chunk you would need to achieve 90% of peak bandwidth (turns out it is about 3.6MB), or even 99% of peak bandwidth (39.6MB!). As you can see, the closer you want to get to peak, the bigger these chunks get (see Figure 41.2 for a plot of these values).

FFS did not use this type of calculation in order to spread large files across groups, however. Instead, it took a simple approach, based on the

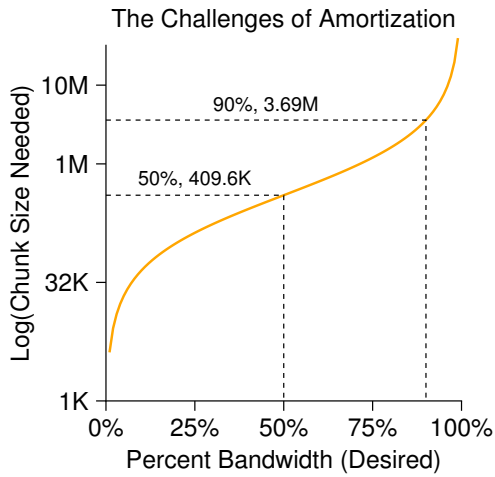


Figure 41.2: **Amortization: How Big Do Chunks Have To Be?**

structure of the inode itself. The first twelve direct blocks were placed in the same group as the inode; each subsequent indirect block, and all the blocks it pointed to, was placed in a different group. With a block size of 4KB, and 32-bit disk addresses, this strategy implies that every 1024 blocks of the file (4MB) were placed in separate groups, the lone exception being the first 48KB of the file as pointed to by direct pointers.

Note that the trend in disk drives is that transfer rate improves fairly rapidly, as disk manufacturers are good at cramming more bits into the same surface, but the mechanical aspects of drives related to seeks (disk arm speed and the rate of rotation) improve rather slowly [P98]. The implication is that over time, mechanical costs become relatively more expensive, and thus, to amortize said costs, you have to transfer more data between seeks.

41.7 A Few Other Things About FFS

FFS introduced a few other innovations too. In particular, the designers were extremely worried about accommodating small files; as it turned out, many files were 2KB or so in size back then, and using 4KB blocks, while good for transferring data, was not so good for space efficiency. This **internal fragmentation** could thus lead to roughly half the disk being wasted for a typical file system.

The solution the FFS designers hit upon was simple and solved the problem. They decided to introduce **sub-blocks**, which were 512-byte little blocks that the file system could allocate to files. Thus, if you created a small file (say 1KB in size), it would occupy two sub-blocks and thus not

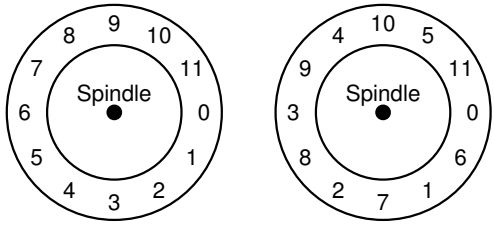


Figure 41.3: **FFS: Standard Versus Parameterized Placement**

waste an entire 4KB block. As the file grew, the file system will continue allocating 512-byte blocks to it until it acquires a full 4KB of data. At that point, FFS will find a 4KB block, *copy* the sub-blocks into it, and free the sub-blocks for future use.

You might observe that this process is inefficient, requiring a lot of extra work for the file system (in particular, a lot of extra I/O to perform the copy). And you'd be right again! Thus, FFS generally avoided this pessimal behavior by modifying the `libc` library; the library would buffer writes and then issue them in 4KB chunks to the file system, thus avoiding the sub-block specialization entirely in most cases.

A second neat thing that FFS introduced was a disk layout that was optimized for performance. In those times (before SCSI and other more modern device interfaces), disks were much less sophisticated and required the host CPU to control their operation in a more hands-on way. A problem arose in FFS when a file was placed on consecutive sectors of the disk, as on the left in Figure 41.3.

In particular, the problem arose during sequential reads. FFS would first issue a read to block 0; by the time the read was complete, and FFS issued a read to block 1, it was too late: block 1 had rotated under the head and now the read to block 1 would incur a full rotation.

FFS solved this problem with a different layout, as you can see on the right in Figure 41.3. By skipping over every other block (in the example), FFS has enough time to request the next block before it went past the disk head. In fact, FFS was smart enough to figure out for a particular disk *how many* blocks it should skip in doing layout in order to avoid the extra rotations; this technique was called **parameterization**, as FFS would figure out the specific performance parameters of the disk and use those to decide on the exact staggered layout scheme.

You might be thinking: this scheme isn't so great after all. In fact, you will only get 50% of peak bandwidth with this type of layout, because you have to go around each track twice just to read each block once. Fortunately, modern disks are much smarter: they internally read the entire track in and buffer it in an internal disk cache (often called a **track buffer** for this very reason). Then, on subsequent reads to the track, the disk will

TIP: MAKE THE SYSTEM USABLE
Probably the most basic lesson from FFS is that not only did it introduce the conceptually good idea of disk-aware layout, but it also added a number of features that simply made the system more usable. Long file names, symbolic links, and a rename operation that worked atomically all improved the utility of a system; while hard to write a research paper about (imagine trying to read a 14-pager about "The Symbolic Link: Hard Link's Long Lost Cousin"), such small features made FFS more useful and thus likely increased its chances for adoption. Making a system usable is often as or more important than its deep technical innovations.

just return the desired data from its cache. File systems thus no longer have to worry about these incredibly low-level details. Abstraction and higher-level interfaces can be a good thing, when designed properly.

Some other usability improvements were added as well. FFS was one of the first file systems to allow for **long file names**, thus enabling more expressive names in the file system instead of the traditional fixed-size approach (e.g., 8 characters). Further, a new concept was introduced called a **symbolic link**. As discussed in a previous chapter [AD14b] , hard links are limited in that they both could not point to directories (for fear of introducing loops in the file system hierarchy) and that they can only point to files within the same volume (i.e., the inode number must still be meaningful). Symbolic links allow the user to create an "alias" to any other file or directory on a system and thus are much more flexible. FFS also introduced an atomic `rename()` operation for renaming files. Usability improvements, beyond the basic technology, also likely gained FFS a stronger user base.

41.8 Summary

The introduction of FFS was a watershed moment in file system history, as it made clear that the problem of file management was one of the most interesting issues within an operating system, and showed how one might begin to deal with that most important of devices, the hard disk. Since that time, hundreds of new file systems have developed, but still today many file systems take cues from FFS (e.g., Linux ext2 and ext3 are obvious intellectual descendants). Certainly all modern systems account for the main lesson of FFS: treat the disk like it's a disk.

References

[AD14a] "Operating Systems: Three Easy Pieces" (Chapter: Hard Disk Drives) by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. Arpaci-Dusseau Books, 2014. *There is no way you should be reading about FFS without having first understood hard drives in some detail. If you try to do so, please instead go directly to jail; do not pass go, and, critically, do not collect 200 much-needed simoleons.*

[AD14b] "Operating Systems: Three Easy Pieces" (Chapter: File System Implementation) by Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau . Arpaci-Dusseau Books, 2014. *As above, it makes little sense to read this chapter unless you have read (and understood) the chapter on file system implementation. Otherwise, we'll be throwing around terms like "inode" and "indirect block" and you'll be like "huh?" and that is no fun for either of us.*

[K94] "The Design of the SEER Predictive Caching System" by G. H. Kuenning. MOBICOMM '94, Santa Cruz, California, December 1994. *According to Kuenning, this is the best overview of the SEER project, which led to (among other things) the collection of these traces.*

[MJLF84] "A Fast File System for UNIX" by Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry. ACM TOCS, 2:3, August 1984. *McKusick was recently honored with the IEEE Reynold B. Johnson award for his contributions to file systems, much of which was based on his work building FFS. In his acceptance speech, he discussed the original FFS software: only 1200 lines of code! Modern versions are a little more complex, e.g., the BSD FFS descendant now is in the 50-thousand lines-of-code range.*

[P98] "Hardware Technology Trends and Database Opportunities" by David A. Patterson. Keynote Lecture at SIGMOD '98, June 1998. *A great and simple overview of disk technology trends and how they change over time.*

Homework (Simulation)

This section introduces `ffs.py`, a simple FFS simulator you can use to understand better how FFS-based file and directory allocation work. See the README for details on how to run the simulator.

Questions

1. Examine the file `in.largefile`, and then run the simulator with flag `-f in.largefile` and `-L 4`. The latter sets the large-file exception to 4 blocks. What will the resulting allocation look like? Run with `-c` to check.
2. Now run with `-L 30`. What do you expect to see? Once again, turn on `-c` to see if you were right. You can also use `-S` to see exactly which blocks were allocated to the file `/a`.
3. Now we will compute some statistics about the file. The first is something we call *filespan*, which is the max distance between any two data blocks of the file or between the inode and any data block. Calculate the filespan of `/a`. Run `ffs.py -f in.largefile -L 4 -T -c` to see what it is. Do the same with `-L 100`. What difference do you expect in filespan as the large-file exception parameter changes from low values to high values?
4. Now let's look at a new input file, `in.manyfiles`. How do you think the FFS policy will lay these files out across groups? (you can run with `-v` to see what files and directories are created, or just `cat in.manyfiles`). Run the simulator with `-c` to see if you were right.
5. A metric to evaluate FFS is called *dirspan*. This metric calculates the spread of files within a particular directory, specifically the max distance between the inodes and data blocks of all files in the directory and the inode and data block of the directory itself. Run with `in.manyfiles` and the `-T` flag, and calculate the dirspan of the three directories. Run with `-c` to check. How good of a job does FFS do in minimizing dirspan?
6. Now change the size of the inode table per group to 5 (`-i 5`). How do you think this will change the layout of the files? Run with `-c` to see if you were right. How does it affect the dirspan?
7. Which group should FFS place inode of a new directory in? The default (simulator) policy looks for the group with the most free inodes. A different policy looks for a set of groups with the most free inodes. For example, if you run with `-A 2`, when allocating a new directory, the simulator will look at groups in pairs and pick the best pair for the allocation. Run `./ffs.py -f in.manyfiles -i 5 -A 2 -c` to see how allocation changes with this strategy. How does it affect dirspan? Why might this policy be good?
8. One last policy change we will explore relates to file fragmentation. Run `./ffs.py -f in.fragmented -v` and see if you can predict how the files that remain are allocated. Run with `-c` to confirm your answer. What is interesting about the data layout of file `/i`? Why is it problematic?
9. A new policy, which we call *contiguous allocation* (`-C`), tries to ensure that each file is allocated contiguously. Specifically, with `-C n`, the file system tries to ensure that `n` contiguous blocks are free within a group before allocating a block. Run `./ffs.py -f in.fragmented -v -C 2 -c` to see the difference. How does layout change as the parameter passed to `-C` increases? Finally, how does `-C` affect filespan and dirspan?

42

Crash Consistency: FSCK and Journaling

As we've seen thus far, the file system manages a set of data structures to implement the expected abstractions: files, directories, and all of the other metadata needed to support the basic abstraction that we expect from a file system. Unlike most data structures (for example, those found in memory of a running program), file system data structures must **persist**, i.e., they must survive over the long haul, stored on devices that retain data despite power loss (such as hard disks or flash-based SSDs).

One major challenge faced by a file system is how to update persistent data structures despite the presence of a **power loss** or **system crash**. Specifically, what happens if, right in the middle of updating on-disk structures, someone trips over the power cord and the machine loses power? Or the operating system encounters a bug and crashes? Because of power losses and crashes, updating a persistent data structure can be quite tricky, and leads to a new and interesting problem in file system implementation, known as the **crash-consistency problem**.

This problem is quite simple to understand. Imagine you have to update two on-disk structures, A and B, in order to complete a particular operation. Because the disk only services a single request at a time, one of these requests will reach the disk first (either A or B). If the system crashes or loses power after one write completes, the on-disk structure will be left in an **inconsistent** state. And thus, we have a problem that all file systems need to solve:

THE CRUX: HOW TO UPDATE THE DISK DESPITE CRASHES

The system may crash or lose power between any two writes, and thus the on-disk state may only partially get updated. After the crash, the system boots and wishes to mount the file system again (in order to access files and such). Given that crashes can occur at arbitrary points in time, how do we ensure the file system keeps the on-disk image in a reasonable state?

In this chapter, we'll describe this problem in more detail, and look at some methods file systems have used to overcome it. We'll begin by examining the approach taken by older file systems, known as **fsck** or the **file system checker**. We'll then turn our attention to another approach, known as **journaling** (also known as **write-ahead logging**), a technique which adds a little bit of overhead to each write but recovers more quickly from crashes or power losses. We will discuss the basic machinery of journaling, including a few different flavors of journaling that Linux ext3 [T98,PAA05] (a relatively modern journaling file system) implements.

42.1 A Detailed Example

To kick off our investigation of journaling, let's look at an example. We'll need to use a **workload** that updates on-disk structures in some way. Assume here that the workload is simple: the append of a single data block to an existing file. The append is accomplished by opening the file, calling `lseek()` to move the file offset to the end of the file, and then issuing a single 4KB write to the file before closing it.

Let's also assume we are using standard simple file system structures on the disk, similar to file systems we have seen before. This tiny example includes an **inode bitmap** (with just 8 bits, one per inode), a **data bitmap** (also 8 bits, one per data block), inodes (8 total, numbered 0 to 7, and spread across four blocks), and data blocks (8 total, numbered 0 to 7). Here is a diagram of this file system:

If you look at the structures in the picture, you can see that a single inode is allocated (inode number 2), which is marked in the inode bitmap, and a single allocated data block (data block 4), also marked in the data bitmap. The inode is denoted I[v1], as it is the first version of this inode; it will soon be updated (due to the workload described above).

Let's peek inside this simplified inode too. Inside of I[v1], we see:

```
owner       : remzi
permissions : read-write
size        : 1
pointer     : 4
pointer     : null
pointer     : null
pointer     : null
```

In this simplified inode, the `size` of the file is `1` (it has one block allocated), the first direct pointer points to block `4` (the first data block of

the file, Da), and all three other direct pointers are set to `null` (indicating that they are not used). Of course, real inodes have many more fields; see previous chapters for more information.

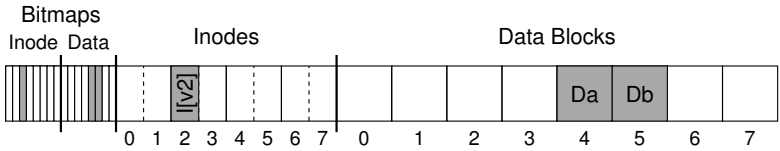
When we append to the file, we are adding a new data block to it, and thus must update three on-disk structures: the inode (which must point to the new block and record the new larger size due to the append), the new data block Db, and a new version of the data bitmap (call it B[v2]) to indicate that the new data block has been allocated.

Thus, in the memory of the system, we have three blocks which we must write to disk. The updated inode (inode version 2, or I[v2] for short) now looks like this:

```
owner       : remzi
permissions : read-write
size        : 2
pointer     : 4
pointer     : 5
pointer     : null
pointer     : null
```

The updated data bitmap (B[v2]) now looks like this: 00001100. Finally, there is the data block (Db), which is just filled with whatever it is users put into files. Stolen music, perhaps?

What we would like is for the final on-disk image of the file system to look like this:



To achieve this transition, the file system must perform three separate writes to the disk, one each for the inode (I[v2]), bitmap (B[v2]), and data block (Db). Note that these writes usually don't happen immediately when the user issues a `write()` system call; rather, the dirty inode, bitmap, and new data will sit in main memory (in the **page cache** or **buffer cache**) for some time first; then, when the file system finally decides to write them to disk (after say 5 seconds or 30 seconds), the file system will issue the requisite write requests to the disk. Unfortunately, a crash may occur and thus interfere with these updates to the disk. In particular, if a crash happens after one or two of these writes have taken place, but not all three, the file system could be left in a funny state.

Crash Scenarios

To understand the problem better, let's look at some example crash scenarios. Imagine only a single write succeeds; there are thus three possible outcomes, which we list here:

- **Just the data block (Db) is written to disk.** In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This case is not a problem at all, from the perspective of file-system crash consistency[1].

- **Just the updated inode (I[v2]) is written to disk.** In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read **garbage** data from the disk (the old contents of disk address 5).

  Further, we have a new problem, which we call a **file-system inconsistency**. The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has. The disagreement between the bitmap and the inode is an inconsistency in the data structures of the file system; to use the file system, we must somehow resolve this problem (more on that below).

- **Just the updated bitmap (B[v2]) is written to disk.** In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again; if left unresolved, this write would result in a **space leak**, as block 5 would never be used by the file system.

There are also three more crash scenarios in this attempt to write three blocks to disk. In these cases, two writes succeed and the last one fails:

- **The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db).** In this case, the file system metadata is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.

- **The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2]).** In this case, we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we once again need to resolve the problem before using the file system.

- **The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2]).** In this case, we again have an inconsistency between the inode and the data bitmap. However, even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file.

[1] However, it might be a problem for the user, who just lost some data!

The Crash Consistency Problem

Hopefully, from these crash scenarios, you can see the many problems that can occur to our on-disk file system image because of crashes: we can have inconsistency in file system data structures; we can have space leaks; we can return garbage data to a user; and so forth. What we'd like to do ideally is move the file system from one consistent state (e.g., before the file got appended to) to another **atomically** (e.g., after the inode, bitmap, and new data block have been written to disk). Unfortunately, we can't do this easily because the disk only commits one write at a time, and crashes or power loss may occur between these updates. We call this general problem the **crash-consistency problem** (we could also call it the **consistent-update problem**).

42.2 Solution #1: The File System Checker

Early file systems took a simple approach to crash consistency. Basically, they decided to let inconsistencies happen and then fix them later (when rebooting). A classic example of this lazy approach is found in a tool that does this: **fsck**[2]. `fsck` is a UNIX tool for finding such inconsistencies and repairing them [MK96]; similar tools to check and repair a disk partition exist on different systems. Note that such an approach can't fix all problems; consider, for example, the case above where the file system looks consistent but the inode points to garbage data. The only real goal is to make sure the file system metadata is internally consistent.

The tool `fsck` operates in a number of phases, as summarized in McKusick and Kowalski's paper [MK96]. It is run *before* the file system is mounted and made available (`fsck` assumes that no other file-system activity is on-going while it runs); once finished, the on-disk file system should be consistent and thus can be made accessible to users.

Here is a basic summary of what `fsck` does:

- **Superblock:** `fsck` first checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks that have been allocated. Usually the goal of these sanity checks is to find a suspect (corrupt) superblock; in this case, the system (or administrator) may decide to use an alternate copy of the superblock.
- **Free blocks:** Next, `fsck` scans the inodes, indirect blocks, double indirect blocks, etc., to build an understanding of which blocks are currently allocated within the file system. It uses this knowledge to produce a correct version of the allocation bitmaps; thus, if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes. The same type of check is performed for all the inodes, making sure that all inodes that look like they are in use are marked as such in the inode bitmaps.

[2] Pronounced either "eff-ess-see-kay", "eff-ess-check", or, if you don't like the tool, "eff-suck". Yes, serious professional people use this term.

- **Inode state:** Each inode is checked for corruption or other problems. For example, `fsck` makes sure that each allocated inode has a valid type field (e.g., regular file, directory, symbolic link, etc.). If there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by `fsck`; the inode bitmap is correspondingly updated.
- **Inode links:** `fsck` also verifies the link count of each allocated inode. As you may recall, the link count indicates the number of different directories that contain a reference (i.e., a link) to this particular file. To verify the link count, `fsck` scans through the entire directory tree, starting at the root directory, and builds its own link counts for every file and directory in the file system. If there is a mismatch between the newly-calculated count and that found within an inode, corrective action must be taken, usually by fixing the count within the inode. If an allocated inode is discovered but no directory refers to it, it is moved to the `lost+found` directory.
- **Duplicates:** `fsck` also checks for duplicate pointers, i.e., cases where two different inodes refer to the same block. If one inode is obviously bad, it may be cleared. Alternately, the pointed-to block could be copied, thus giving each inode its own copy as desired.
- **Bad blocks:** A check for bad block pointers is also performed while scanning through the list of all pointers. A pointer is considered "bad" if it obviously points to something outside its valid range, e.g., it has an address that refers to a block greater than the partition size. In this case, `fsck` can't do anything too intelligent; it just removes (clears) the pointer from the inode or indirect block.
- **Directory checks:** `fsck` does not understand the contents of user files; however, directories hold specifically formatted information created by the file system itself. Thus, `fsck` performs additional integrity checks on the contents of each directory, making sure that "." and ".." are the first entries, that each inode referred to in a directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.

As you can see, building a working `fsck` requires intricate knowledge of the file system; making sure such a piece of code works correctly in all cases can be challenging [G+08]. However, fsck (and similar approaches) have a bigger and perhaps more fundamental problem: they are *too slow*. With a very large disk volume, scanning the entire disk to find all the allocated blocks and read the entire directory tree may take many minutes or hours. Performance of `fsck`, as disks grew in capacity and RAIDs grew in popularity, became prohibitive (despite recent advances [M+13]).

At a higher level, the basic premise of `fsck` seems just a tad irrational. Consider our example above, where just three blocks are written to the disk; it is incredibly expensive to scan the entire disk to fix problems that occurred during an update of just three blocks. This situation is akin to dropping your keys on the floor in your bedroom, and then com-

mencing a *search-the-entire-house-for-keys* recovery algorithm, starting in the basement and working your way through every room. It works but is wasteful. Thus, as disks (and RAIDs) grew, researchers and practitioners started to look for other solutions.

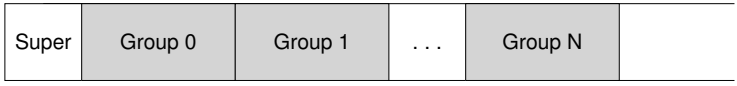
42.3 Solution #2: Journaling (or Write-Ahead Logging)

Probably the most popular solution to the consistent update problem is to steal an idea from the world of database management systems. That idea, known as **write-ahead logging**, was invented to address exactly this type of problem. In file systems, we usually call write-ahead logging **journaling** for historical reasons. The first file system to do this was Cedar [H87], though many modern file systems use the idea, including Linux ext3 and ext4, reiserfs, IBM's JFS, SGI's XFS, and Windows NTFS.

The basic idea is as follows. When updating the disk, before overwriting the structures in place, first write down a little note (somewhere else on the disk, in a well-known location) describing what you are about to do. Writing this note is the "write ahead" part, and we write it to a structure that we organize as a "log"; hence, write-ahead logging.

By writing the note to disk, you are guaranteeing that if a crash takes places during the update (overwrite) of the structures you are updating, you can go back and look at the note you made and try again; thus, you will know exactly what to fix (and how to fix it) after a crash, instead of having to scan the entire disk. By design, journaling thus adds a bit of work during updates to greatly reduce the amount of work required during recovery.

We'll now describe how **Linux ext3**, a popular journaling file system, incorporates journaling into the file system. Most of the on-disk structures are identical to **Linux ext2**, e.g., the disk is divided into block groups, and each block group contains an inode bitmap, data bitmap, inodes, and data blocks. The new key structure is the journal itself, which occupies some small amount of space within the partition or on another device. Thus, an ext2 file system (without journaling) looks like this:



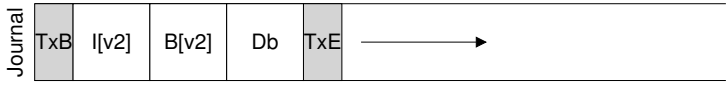
Assuming the journal is placed within the same file system image (though sometimes it is placed on a separate device, or as a file within the file system), an ext3 file system with a journal looks like this:

The real difference is just the presence of the journal, and of course, how it is used.

Data Journaling

Let's look at a simple example to understand how **data journaling** works. Data journaling is available as a mode with the Linux ext3 file system, from which much of this discussion is based.

Say we have our canonical update again, where we wish to write the inode (I[v2]), bitmap (B[v2]), and data block (Db) to disk again. Before writing them to their final disk locations, we are now first going to write them to the log (a.k.a. journal). This is what this will look like in the log:



You can see we have written five blocks here. The transaction begin (TxB) tells us about this update, including information about the pending update to the file system (e.g., the final addresses of the blocks I[v2], B[v2], and Db), and some kind of **transaction identifier** (**TID**). The middle three blocks just contain the exact contents of the blocks themselves; this is known as **physical logging** as we are putting the exact physical contents of the update in the journal (an alternate idea, **logical logging**, puts a more compact logical representation of the update in the journal, e.g., "this update wishes to append data block Db to file X", which is a little more complex but can save space in the log and perhaps improve performance). The final block (TxE) is a marker of the end of this transaction, and will also contain the TID.

Once this transaction is safely on disk, we are ready to overwrite the old structures in the file system; this process is called **checkpointing**. Thus, to **checkpoint** the file system (i.e., bring it up to date with the pending update in the journal), we issue the writes I[v2], B[v2], and Db to their disk locations as seen above; if these writes complete successfully, we have successfully checkpointed the file system and are basically done. Thus, our initial sequence of operations:

1. **Journal write:** Write the transaction, including a transaction-begin block, all pending data and metadata updates, and a transaction-end block, to the log; wait for these writes to complete.
2. **Checkpoint:** Write the pending metadata and data updates to their final locations in the file system.

In our example, we would write TxB, I[v2], B[v2], Db, and TxE to the journal first. When these writes complete, we would complete the update by checkpointing I[v2], B[v2], and Db, to their final locations on disk.

Things get a little trickier when a crash occurs during the writes to the journal. Here, we are trying to write the set of blocks in the transaction (e.g., TxB, I[v2], B[v2], Db, TxE) to disk. One simple way to do this would be to issue each one at a time, waiting for each to complete, and then issuing the next. However, this is slow. Ideally, we'd like to issue

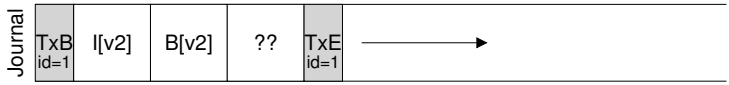
ASIDE: FORCING WRITES TO DISK

To enforce ordering between two disk writes, modern file systems have to take a few extra precautions. In olden times, forcing ordering between two writes, A and B, was easy: just issue the write of A to the disk, wait for the disk to interrupt the OS when the write is complete, and then issue the write of B.

Things got slightly more complex due to the increased use of write caches within disks. With write buffering enabled (sometimes called **immediate reporting**), a disk will inform the OS the write is complete when it simply has been placed in the disk's memory cache, and has not yet reached disk. If the OS then issues a subsequent write, it is not guaranteed to reach the disk after previous writes; thus ordering between writes is not preserved. One solution is to disable write buffering. However, more modern systems take extra precautions and issue explicit **write barriers**; such a barrier, when it completes, guarantees that all writes issued before the barrier will reach disk before any writes issued after the barrier.

All of this machinery requires a great deal of trust in the correct operation of the disk. Unfortunately, recent research shows that some disk manufacturers, in an effort to deliver "higher performing" disks, explicitly ignore write-barrier requests, thus making the disks seemingly run faster but at the risk of incorrect operation [C+13, R+11]. As Kahan said, the fast almost always beats out the slow, even if the fast is wrong.

all five block writes at once, as this would turn five writes into a single sequential write and thus be faster. However, this is unsafe, for the following reason: given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order. Thus, the disk internally may (1) write TxB, I[v2], B[v2], and TxE and only later (2) write Db. Unfortunately, if the disk loses power between (1) and (2), this is what ends up on disk:



Why is this a problem? Well, the transaction looks like a valid transaction (it has a begin and an end with matching sequence numbers). Further, the file system can't look at that fourth block and know it is wrong; after all, it is arbitrary user data. Thus, if the system now reboots and runs recovery, it will replay this transaction, and ignorantly copy the contents of the garbage block '??' to the location where Db is supposed to live. This is bad for arbitrary user data in a file; it is much worse if it happens to a critical piece of file system, such as the superblock, which could render the file system unmountable.

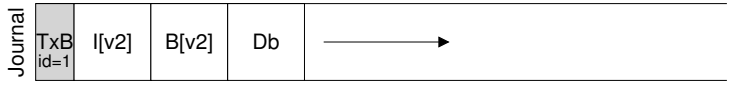
ASIDE: OPTIMIZING LOG WRITES

You may have noticed a particular inefficiency of writing to the log. Namely, the file system first has to write out the transaction-begin block and contents of the transaction; only after these writes complete can the file system send the transaction-end block to disk. The performance impact is clear, if you think about how a disk works: usually an extra rotation is incurred (think about why).

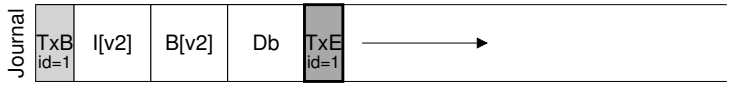
One of our former graduate students, Vijayan Prabhakaran, had a simple idea to fix this problem [P+05]. When writing a transaction to the journal, include a checksum of the contents of the journal in the begin and end blocks. Doing so enables the file system to write the entire transaction at once, without incurring a wait; if, during recovery, the file system sees a mismatch in the computed checksum versus the stored checksum in the transaction, it can conclude that a crash occurred during the write of the transaction and thus discard the file-system update. Thus, with a small tweak in the write protocol and recovery system, a file system can achieve faster common-case performance; on top of that, the system is slightly more reliable, as any reads from the journal are now protected by a checksum.

This simple fix was attractive enough to gain the notice of Linux file system developers, who then incorporated it into the next generation Linux file system, called (you guessed it!) **Linux ext4**. It now ships on millions of machines worldwide, including the Android handheld platform. Thus, every time you write to disk on many Linux-based systems, a little code developed at Wisconsin makes your system a little faster and more reliable.

To avoid this problem, the file system issues the transactional write in two steps. First, it writes all blocks except the TxE block to the journal, issuing these writes all at once. When these writes complete, the journal will look something like this (assuming our append workload again):



When those writes complete, the file system issues the write of the TxE block, thus leaving the journal in this final, safe state:



An important aspect of this process is the atomicity guarantee provided by the disk. It turns out that the disk guarantees that any 512-byte

write will either happen or not (and never be half-written); thus, to make sure the write of TxE is atomic, one should make it a single 512-byte block. Thus, our current protocol to update the file system, with each of its three phases labeled:

1. **Journal write:** Write the contents of the transaction (including TxB, metadata, and data) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for write to complete; transaction is said to be **committed**.
3. **Checkpoint:** Write the contents of the update (metadata and data) to their final on-disk locations.

Recovery

Let's now understand how a file system can use the contents of the journal to **recover** from a crash. A crash may happen at any time during this sequence of updates. If the crash happens before the transaction is written safely to the log (i.e., before Step 2 above completes), then our job is easy: the pending update is simply skipped. If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can **recover** the update as follows. When the system boots, the file system recovery process will scan the log and look for transactions that have committed to the disk; these transactions are thus **replayed** (in order), with the file system again attempting to write out the blocks in the transaction to their final on-disk locations. This form of logging is one of the simplest forms there is, and is called **redo logging**. By recovering the committed transactions in the journal, the file system ensures that the on-disk structures are consistent, and thus can proceed by mounting the file system and readying itself for new requests.

Note that it is fine for a crash to happen at any point during checkpointing, even after some of the updates to the final locations of the blocks have completed. In the worst case, some of these updates are simply performed again during recovery. Because recovery is a rare operation (only taking place after an unexpected system crash), a few redundant writes are nothing to worry about[3].

Batching Log Updates

You might have noticed that the basic protocol could add a lot of extra disk traffic. For example, imagine we create two files in a row, called `file1` and `file2`, in the same directory. To create one file, one has to update a number of on-disk structures, minimally including: the inode bitmap (to allocate a new inode), the newly-created inode of the file,

[3]Unless you worry about everything, in which case we can't help you. Stop worrying so much, it is unhealthy! But now you're probably worried about over-worrying.

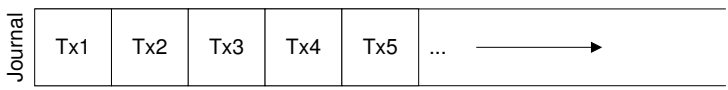
the data block of the parent directory containing the new directory entry, and the parent directory inode (which now has a new modification time). With journaling, we logically commit all of this information to the journal for each of our two file creations; because the files are in the same directory, and assuming they even have inodes within the same inode block, this means that if we're not careful, we'll end up writing these same blocks over and over.

To remedy this problem, some file systems do not commit each update to disk one at a time (e.g., Linux ext3); rather, one can buffer all updates into a global transaction. In our example above, when the two files are created, the file system just marks the in-memory inode bitmap, inodes of the files, directory data, and directory inode as dirty, and adds them to the list of blocks that form the current transaction. When it is finally time to write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates described above. Thus, by buffering updates, a file system can avoid excessive write traffic to disk in many cases.

Making The Log Finite

We thus have arrived at a basic protocol for updating file-system on-disk structures. The file system buffers updates in memory for some time; when it is finally time to write to disk, the file system first carefully writes out the details of the transaction to the journal (a.k.a. write-ahead log); after the transaction is complete, the file system checkpoints those blocks to their final locations on disk.

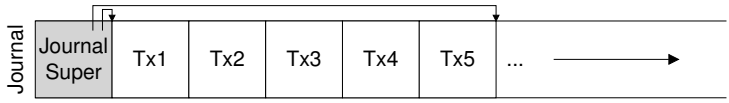
However, the log is of a finite size. If we keep adding transactions to it (as in this figure), it will soon fill. What do you think happens then?



Two problems arise when the log becomes full. The first is simpler, but less critical: the larger the log, the longer recovery will take, as the recovery process must replay all the transactions within the log (in order) to recover. The second is more of an issue: when the log is full (or nearly full), no further transactions can be committed to the disk, thus making the file system "less than useful" (i.e., useless).

To address these problems, journaling file systems treat the log as a circular data structure, re-using it over and over; this is why the journal is sometimes referred to as a **circular log**. To do so, the file system must take action some time after a checkpoint. Specifically, once a transaction has been checkpointed, the file system should free the space it was occupying within the journal, allowing the log space to be reused. There are many ways to achieve this end; for example, you could simply mark the

oldest and newest non-checkpointed transactions in the log in a **journal superblock**; all other space is free. Here is a graphical depiction:



In the journal superblock (not to be confused with the main file system superblock), the journaling system records enough information to know which transactions have not yet been checkpointed, and thus reduces recovery time as well as enables re-use of the log in a circular fashion. And thus we add another step to our basic protocol:

1. **Journal write:** Write the contents of the transaction (containing TxB and the contents of the update) to the log; wait for these writes to complete.
2. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction is now **committed**.
3. **Checkpoint:** Write the contents of the update to their final locations within the file system.
4. **Free:** Some time later, mark the transaction free in the journal by updating the journal superblock.

Thus we have our final data journaling protocol. But there is still a problem: we are writing each data block to the disk *twice*, which is a heavy cost to pay, especially for something as rare as a system crash. Can you figure out a way to retain consistency without writing data twice?

Metadata Journaling

Although recovery is now fast (scanning the journal and replaying a few transactions as opposed to scanning the entire disk), normal operation of the file system is slower than we might desire. In particular, for each write to disk, we are now also writing to the journal first, thus doubling write traffic; this doubling is especially painful during sequential write workloads, which now will proceed at half the peak write bandwidth of the drive. Further, between writes to the journal and writes to the main file system, there is a costly seek, which adds noticeable overhead for some workloads.

Because of the high cost of writing every data block to disk twice, people have tried a few different things in order to speed up performance. For example, the mode of journaling we described above is often called **data journaling** (as in Linux ext3), as it journals all user data (in addition to the metadata of the file system). A simpler (and more common) form of journaling is sometimes called **ordered journaling** (or just **metadata**

journaling), and it is nearly the same, except that user data is *not* written to the journal. Thus, when performing the same update as above, the following information would be written to the journal:

The data block Db, previously written to the log, would instead be written to the file system proper, avoiding the extra write; given that most I/O traffic to the disk is data, not writing data twice substantially reduces the I/O load of journaling. The modification does raise an interesting question, though: when should we write data blocks to disk?

Let's again consider our example append of a file to understand the problem better. The update consists of three blocks: I[v2], B[v2], and Db. The first two are both metadata and will be logged and then checkpointed; the latter will only be written once to the file system. When should we write Db to disk? Does it matter?

As it turns out, the ordering of the data write does matter for metadata-only journaling. For example, what if we write Db to disk *after* the transaction (containing I[v2] and B[v2]) completes? Unfortunately, this approach has a problem: the file system is consistent but I[v2] may end up pointing to garbage data. Specifically, consider the case where I[v2] and B[v2] are written but Db did not make it to disk. The file system will then try to recover. Because Db is *not* in the log, the file system will replay writes to I[v2] and B[v2], and produce a consistent file system (from the perspective of file-system metadata). However, I[v2] will be pointing to garbage data, i.e., at whatever was in the slot where Db was headed.

To ensure this situation does not arise, some file systems (e.g., Linux ext3) write data blocks (of regular files) to the disk *first*, before related metadata is written to disk. Specifically, the protocol is as follows:

1. **Data write:** Write data to final location; wait for completion (the wait is optional; see below for details).
2. **Journal metadata write:** Write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit:** Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now **committed**.
4. **Checkpoint metadata:** Write the contents of the metadata update to their final locations within the file system.
5. **Free:** Later, mark the transaction free in journal superblock.

By forcing the data write first, a file system can guarantee that a pointer will never point to garbage. Indeed, this rule of "write the pointed-to object before the object that points to it" is at the core of crash consistency, and is exploited even further by other crash consistency schemes [GP94] (see below for details).

In most systems, metadata journaling (akin to ordered journaling of ext3) is more popular than full data journaling. For example, Windows NTFS and SGI's XFS both use some form of metadata journaling. Linux ext3 gives you the option of choosing either data, ordered, or unordered modes (in unordered mode, data can be written at any time). All of these modes keep metadata consistent; they vary in their semantics for data.

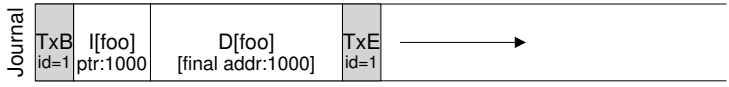
Finally, note that forcing the data write to complete (Step 1) before issuing writes to the journal (Step 2) is not required for correctness, as indicated in the protocol above. Specifically, it would be fine to concurrently issue writes to data, the transaction-begin block, and journaled metadata; the only real requirement is that Steps 1 and 2 complete before the issuing of the journal commit block (Step 3).

Tricky Case: Block Reuse

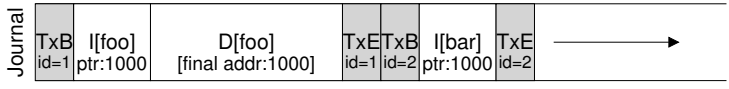
There are some interesting corner cases that make journaling more tricky, and thus are worth discussing. A number of them revolve around block reuse; as Stephen Tweedie (one of the main forces behind ext3) said:

> "What's the hideous part of the entire system? ... It's deleting files. Everything to do with delete is hairy. Everything to do with delete... you have nightmares around what happens if blocks get deleted and then reallocated." [T00]

The particular example Tweedie gives is as follows. Suppose you are using some form of metadata journaling (and thus data blocks for files are *not* journaled). Let's say you have a directory called `foo`. The user adds an entry to `foo` (say by creating a file), and thus the contents of `foo` (because directories are considered metadata) are written to the log; assume the location of the `foo` directory data is block 1000. The log thus contains something like this:



At this point, the user deletes everything in the directory and the directory itself, freeing up block 1000 for reuse. Finally, the user creates a new file (say `bar`), which ends up reusing the same block (1000) that used to belong to `foo`. The inode of `bar` is committed to disk, as is its data; note, however, because metadata journaling is in use, only the inode of `bar` is committed to the journal; the newly-written data in block 1000 in the file `bar` is *not* journaled.



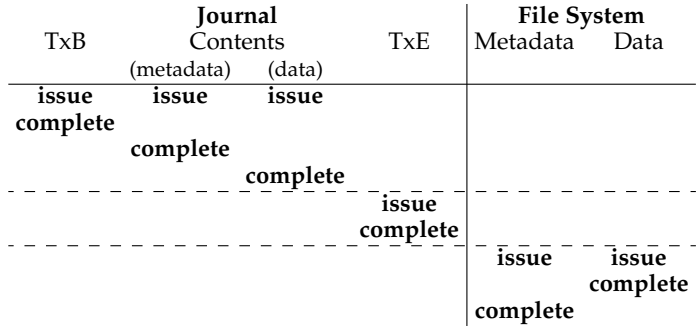


Figure 42.1: **Data Journaling Timeline**

Now assume a crash occurs and all of this information is still in the log. During replay, the recovery process simply replays everything in the log, including the write of directory data in block 1000; the replay thus overwrites the user data of current file `bar` with old directory contents! Clearly this is not a correct recovery action, and certainly it will be a surprise to the user when reading the file `bar`.

There are a number of solutions to this problem. One could, for example, never reuse blocks until the delete of said blocks is checkpointed out of the journal. What Linux ext3 does instead is to add a new type of record to the journal, known as a **revoke** record. In the case above, deleting the directory would cause a revoke record to be written to the journal. When replaying the journal, the system first scans for such revoke records; any such revoked data is never replayed, thus avoiding the problem mentioned above.

Wrapping Up Journaling: A Timeline

Before ending our discussion of journaling, we summarize the protocols we have discussed with timelines depicting each of them. Figure 42.1 shows the protocol when journaling data and metadata, whereas Figure 42.2 shows the protocol when journaling only metadata.

In each figure, time increases in the downward direction, and each row in the figure shows the logical time that a write can be issued or might complete. For example, in the data journaling protocol (Figure 42.1), the writes of the transaction begin block (TxB) and the contents of the transaction can logically be issued at the same time, and thus can be completed in any order; however, the write to the transaction end block (TxE) must not be issued until said previous writes complete. Similarly, the checkpointing writes to data and metadata blocks cannot begin until the transaction end block has committed. Horizontal dashed lines show where write-ordering requirements must be obeyed.

A similar timeline is shown for the metadata journaling protocol. Note

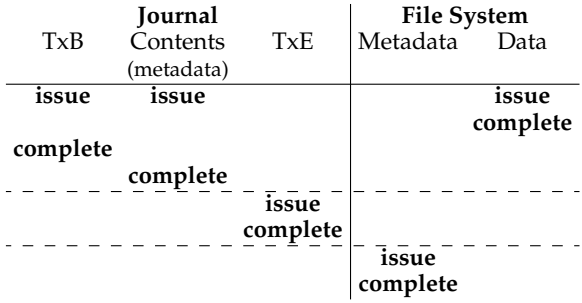


Figure 42.2: **Metadata Journaling Timeline**

that the data write can logically be issued at the same time as the writes to the transaction begin and the contents of the journal; however, it must be issued and complete before the transaction end has been issued.

Finally, note that the time of completion marked for each write in the timelines is arbitrary. In a real system, completion time is determined by the I/O subsystem, which may reorder writes to improve performance. The only guarantees about ordering that we have are those that must be enforced for protocol correctness (and are shown via the horizontal dashed lines in the figures).

42.4 Solution #3: Other Approaches

We've thus far described two options in keeping file system metadata consistent: a lazy approach based on `fsck`, and a more active approach known as journaling. However, these are not the only two approaches. One such approach, known as Soft Updates [GP94], was introduced by Ganger and Patt. This approach carefully orders all writes to the file system to ensure that the on-disk structures are never left in an inconsistent state. For example, by writing a pointed-to data block to disk *before* the inode that points to it, we can ensure that the inode never points to garbage; similar rules can be derived for all the structures of the file system. Implementing Soft Updates can be a challenge, however; whereas the journaling layer described above can be implemented with relatively little knowledge of the exact file system structures, Soft Updates requires intricate knowledge of each file system data structure and thus adds a fair amount of complexity to the system.

Another approach is known as **copy-on-write** (yes, **COW**), and is used in a number of popular file systems, including Sun's ZFS [B07]. This technique never overwrites files or directories in place; rather, it places new updates to previously unused locations on disk. After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures. Doing so makes keeping the file system consistent straightforward. We'll be learn-

ing more about this technique when we discuss the log-structured file system (LFS) in a future chapter; LFS is an early example of a COW.

Another approach is one we just developed here at Wisconsin. In this technique, entitled **backpointer-based consistency** (or **BBC**), no ordering is enforced between writes. To achieve consistency, an additional **back pointer** is added to every block in the system; for example, each data block has a reference to the inode to which it belongs. When accessing a file, the file system can determine if the file is consistent by checking if the forward pointer (e.g., the address in the inode or direct block) points to a block that refers back to it. If so, everything must have safely reached disk and thus the file is consistent; if not, the file is inconsistent, and an error is returned. By adding back pointers to the file system, a new form of lazy crash consistency can be attained [C+12].

Finally, we also have explored techniques to reduce the number of times a journal protocol has to wait for disk writes to complete. Entitled **optimistic crash consistency** [C+13], this new approach issues as many writes to disk as possible by using a generalized form of the **transaction checksum** [P+05], and includes a few other techniques to detect inconsistencies should they arise. For some workloads, these optimistic techniques can improve performance by an order of magnitude. However, to truly function well, a slightly different disk interface is required [C+13].

42.5 Summary

We have introduced the problem of crash consistency, and discussed various approaches to attacking this problem. The older approach of building a file system checker works but is likely too slow to recover on modern systems. Thus, many file systems now use journaling. Journaling reduces recovery time from O(size-of-the-disk-volume) to O(size-of-the-log), thus speeding recovery substantially after a crash and restart. For this reason, many modern file systems use journaling. We have also seen that journaling can come in many different forms; the most commonly used is ordered metadata journaling, which reduces the amount of traffic to the journal while still preserving reasonable consistency guarantees for both file system metadata and user data. In the end, strong guarantees on user data are probably one of the most important things to provide; oddly enough, as recent research has shown, this area remains a work in progress [P+14].

References

[B07] "ZFS: The Last Word in File Systems" by Jeff Bonwick and Bill Moore. Available online: `http://www.ostep.org/Citations/zfs_last.pdf`. *ZFS uses copy-on-write and journaling, actually, as in some cases, logging writes to disk will perform better.*

[C+12] "Consistency Without Ordering" by Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '12, San Jose, California. *A recent paper of ours about a new form of crash consistency based on back pointers. Read it for the exciting details!*

[C+13] "Optimistic Crash Consistency" by Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau . SOSP '13, Nemacolin Woodlands Resort, PA, November 2013. *Our work on a more optimistic and higher performance journaling protocol. For workloads that call `fsync()` a lot, performance can be greatly improved.*

[GP94] "Metadata Update Performance in File Systems" by Gregory R. Ganger and Yale N. Patt. OSDI '94. *A clever paper about using careful ordering of writes as the main way to achieve consistency. Implemented later in BSD-based systems.*

[G+08] "SQCK: A Declarative File System Checker" by Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. OSDI '08, San Diego, California. *Our own paper on a new and better way to build a file system checker using SQL queries. We also show some problems with the existing checker, finding numerous bugs and odd behaviors, a direct result of the complexity of `fsck`.*

[H87] "Reimplementing the Cedar File System Using Logging and Group Commit" by Robert Hagmann. SOSP '87, Austin, Texas, November 1987. *The first work (that we know of) that applied write-ahead logging (a.k.a. journaling) to a file system.*

[M+13] "ffsck: The Fast File System Checker" by Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '13, San Jose, California, February 2013. *A recent paper of ours detailing how to make fsck an order of magnitude faster. Some of the ideas have already been incorporated into the BSD file system checker [MK96] and are deployed today.*

[MK96] "Fsck – The UNIX File System Check Program" by Marshall Kirk McKusick and T. J. Kowalski. Revised in 1996. *Describes the first comprehensive file-system checking tool, the eponymous `fsck`. Written by some of the same people who brought you FFS.*

[MJLF84] "A Fast File System for UNIX" by Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry. ACM Transactions on Computing Systems, Volume 2:3, August 1984. *You already know enough about FFS, right? But come on, it is OK to re-reference important papers.*

[P+14] "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications" by Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. OSDI '14, Broomfield, Colorado, October 2014. *A paper in which we study what file systems guarantee after crashes, and show that applications expect something different, leading to all sorts of interesting problems.*

[P+05] "IRON File Systems" by Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. SOSP '05, Brighton, England, October 2005. *A paper mostly focused on studying how file systems react to disk failures. Towards the end, we introduce a transaction checksum to speed up logging, which was eventually adopted into Linux ext4.*

[PAA05] "Analysis and Evolution of Journaling File Systems" by Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. USENIX '05, Anaheim, California, April 2005. *An early paper we wrote analyzing how journaling file systems work.*

[R+11] "Coerced Cache Eviction and Discreet-Mode Journaling" by Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. DSN '11, Hong Kong, China, June 2011. *Our own paper on the problem of disks that buffer writes in a memory cache instead of forcing them to disk, even when explicitly told not to do that! Our solution to overcome this problem: if you want A to be written to disk before B, first write A, then send a lot of "dummy" writes to disk, hopefully causing A to be forced to disk to make room for them in the cache. A neat if impractical solution.*

[T98] "Journaling the Linux ext2fs File System" by Stephen C. Tweedie. The Fourth Annual Linux Expo, May 1998. *Tweedie did much of the heavy lifting in adding journaling to the Linux ext2 file system; the result, not surprisingly, is called ext3. Some nice design decisions include the strong focus on backwards compatibility, e.g., you can just add a journaling file to an existing ext2 file system and then mount it as an ext3 file system.*

[T00] "EXT3, Journaling Filesystem" by Stephen Tweedie. Talk at the Ottawa Linux Symposium, July 2000. olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html *A transcript of a talk given by Tweedie on ext3.*

[T01] "The Linux ext2 File System" by Theodore Ts'o, June, 2001.. Available online here: `http://e2fsprogs.sourceforge.net/ext2.html`. *A simple Linux file system based on the ideas found in FFS. For a while it was quite heavily used; now it is really just in the kernel as an example of a simple file system.*

Homework (Simulation)

This section introduces `fsck.py`, a simple simulator you can use to better understand how file system corruptions can be detected (and potentially repaired). Please see the associated README for details on how to run the simulator.

Questions

1. First, run `fsck.py -D`; this flag turns off any corruption, and thus you can use it to generate a random file system, and see if you can determine which files and directories are in there. So, go ahead and do that! Use the `-p` flag to see if you were right. Try this for a few different randomly-generated file systems by setting the seed (`-s`) to different values, like 1, 2, and 3.
2. Now, let's introduce a corruption. Run `fsck.py -S 1` to start. Can you see what inconsistency is introduced? How would you fix it in a real file system repair tool? Use `-c` to check if you were right.
3. Change the seed to `-S 3` or `-S 19`; which inconsistency do you see? Use `-c` to check your answer. What is different in these two cases?
4. Change the seed to `-S 5`; which inconsistency do you see? How hard would it be to fix this problem in an automatic way? Use `-c` to check your answer. Then, introduce a similar inconsistency with `-S 38`; is this harder/possible to detect? Finally, use `-S 642`; is this inconsistency detectable? If so, how would you fix the file system?
5. Change the seed to `-S 6` or `-S 13`; which inconsistency do you see? Use `-c` to check your answer. What is the difference across these two cases? What should the repair tool do when encountering such a situation?
6. Change the seed to `-S 9`; which inconsistency do you see? Use `-c` to check your answer. Which piece of information should a check-and-repair tool trust in this case?
7. Change the seed to `-S 15`; which inconsistency do you see? Use `-c` to check your answer. What can a repair tool do in this case? If no repair is possible, how much data is lost?
8. Change the seed to `-S 10`; which inconsistency do you see? Use `-c` to check your answer. Is there redundancy in the file system structure here that can help a repair?
9. Change the seed to `-S 16` and `-S 20`; which inconsistency do you see? Use `-c` to check your answer. How should the repair tool fix the problem?

44

Flash-based SSDs

After decades of hard-disk drive dominance, a new form of persistent storage device has recently gained significance in the world. Generically referred to as **solid-state storage**, such devices have no mechanical or moving parts like hard drives; rather, they are simply built out of transistors, much like memory and processors. However, unlike typical random-access memory (e.g., DRAM), such a **solid-state storage device** (a.k.a., an **SSD**) retains information despite power loss, and thus is an ideal candidate for use in persistent storage of data.

The technology we'll focus on is known as **flash** (more specifically, **NAND-based flash**), which was created by Fujio Masuoka in the 1980s [M+14]. Flash, as we'll see, has some unique properties. For example, to write to a given chunk of it (i.e., a **flash page**), you first have to erase a bigger chunk (i.e., a **flash block**), which can be quite expensive. In addition, writing too often to a page will cause it to **wear out**. These two properties make construction of a flash-based SSD an interesting challenge:

CRUX: HOW TO BUILD A FLASH-BASED SSD

How can we build a flash-based SSD? How can we handle the expensive nature of erasing? How can we build a device that lasts a long time, given that repeated overwrite will wear the device out? Will the march of progress in technology ever cease? Or cease to amaze?

44.1 Storing a Single Bit

Flash chips are designed to store one or more bits in a single transistor; the level of charge trapped within the transistor is mapped to a binary value. In a **single-level cell** (**SLC**) flash, only a single bit is stored within a transistor (i.e., 1 or 0); with a **multi-level cell** (**MLC**) flash, two bits are encoded into different levels of charge, e.g., 00, 01, 10, and 11 are represented by low, somewhat low, somewhat high, and high levels. There is even **triple-level cell** (**TLC**) flash, which encodes 3 bits per cell. Overall, SLC chips achieve higher performance and are more expensive.

TIP: BE CAREFUL WITH TERMINOLOGY
You may have noticed that some terms we have used many times before (blocks, pages) are being used within the context of a flash, but in slightly different ways than before. New terms are not created to make your life harder (although they may be doing just that), but arise because there is no central authority where terminology decisions are made. What is a block to you may be a page to someone else, and vice versa, depending on the context. Your job is simple: to know the appropriate terms within each domain, and use them such that people well-versed in the discipline can understand what you are talking about. It's one of those times where the only solution is simple but sometimes painful: use your memory.

Of course, there are many details as to exactly how such bit-level storage operates, down at the level of device physics. While beyond the scope of this book, you can read more about it on your own [J10].

44.2 From Bits to Banks/Planes

As they say in ancient Greece, storing a single bit (or a few) does not a storage system make. Hence, flash chips are organized into **banks** or **planes** which consist of a large number of cells.

A bank is accessed in two different sized units: **blocks** (sometimes called **erase blocks**), which are typically of size 128 KB or 256 KB, and **pages**, which are a few KB in size (e.g., 4KB). Within each bank there are a large number of blocks; within each block, there are a large number of pages. When thinking about flash, you must remember this new terminology, which is different than the blocks we refer to in disks and RAIDs and the pages we refer to in virtual memory.

Figure 44.1 shows an example of a flash plane with blocks and pages; there are three blocks, each containing four pages, in this simple example. We'll see below why we distinguish between blocks and pages; it turns out this distinction is critical for flash operations such as reading and writing, and even more so for the overall performance of the device. The most important (and weird) thing you will learn is that to write to a page within a block, you first have to erase the entire block; this tricky detail makes building a flash-based SSD an interesting and worthwhile challenge, and the subject of the second-half of the chapter.

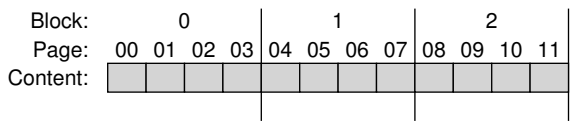


Figure 44.1: **A Simple Flash Chip: Pages Within Blocks**

44.3 Basic Flash Operations

Given this flash organization, there are three low-level operations that a flash chip supports. The **read** command is used to read a page from the flash; **erase** and **program** are used in tandem to write. The details:

- **Read (a page)**: A client of the flash chip can read any page (e.g., 2KB or 4KB), simply by specifying the read command and appropriate page number to the device. This operation is typically quite fast, 10s of microseconds or so, regardless of location on the device, and (more or less) regardless of the location of the previous request (quite unlike a disk). Being able to access any location uniformly quickly means the device is a **random access** device.
- **Erase (a block):** Before writing to a *page* within a flash, the nature of the device requires that you first **erase** the entire *block* the page lies within. Erase, importantly, destroys the contents of the block (by setting each bit to the value 1); therefore, you must be sure that any data you care about in the block has been copied elsewhere (to memory, or perhaps to another flash block) *before* executing the erase. The erase command is quite expensive, taking a few milliseconds to complete. Once finished, the entire block is reset and each page is ready to be programmed.
- **Program (a page):** Once a block has been erased, the program command can be used to change some of the 1's within a page to 0's, and write the desired contents of a page to the flash. Programming a page is less expensive than erasing a block, but more costly than reading a page, usually taking around 100s of microseconds on modern flash chips.

One way to think about flash chips is that each page has a state associated with it. Pages start in an `INVALID` state. By erasing the block that a page resides within, you set the state of the page (and all pages within that block) to `ERASED`, which resets the content of each page in the block but also (importantly) makes them programmable. When you program a page, its state changes to `VALID`, meaning its contents have been set and can be read. Reads do not affect these states (although you should only read from pages that have been programmed). Once a page has been programmed, the only way to change its contents is to erase the entire block within which the page resides. Here is an example of states transition after various erase and program operations within a 4-page block:

		`iiii`	*Initial: pages in block are invalid* (`i`)
Erase()	→	`EEEE`	*State of pages in block set to erased* (`E`)
Program(0)	→	`VEEE`	*Program page 0; state set to valid* (`V`)
Program(0)	→	**error**	*Cannot re-program page after programming*
Program(1)	→	`VVEE`	*Program page 1*
Erase()	→	`EEEE`	*Contents erased; all pages programmable*

A Detailed Example

Because the process of writing (i.e., erasing and programming) is so unusual, let's go through a detailed example to make sure it makes sense. In this example, imagine we have the following four 8-bit pages, within a 4-page block (both unrealistically small sizes, but useful within this example); each page is `VALID` as each has been previously programmed.

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

Now say we wish to write to page 0, filling it with new contents. To write any page, we must first erase the entire block. Let's assume we do so, thus leaving the block in this state:

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

Good news! We could now go ahead and program page 0, for example with the contents `00000011`, overwriting the old page 0 (contents `00011000`) as desired. After doing so, our block looks like this:

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

And now the bad news: the previous contents of pages 1, 2, and 3 are all gone! Thus, before overwriting any page *within* a block, we must first move any data we care about to another location (e.g., memory, or elsewhere on the flash). The nature of erase will have a strong impact on how we design flash-based SSDs, as we'll soon learn about.

Summary

To summarize, reading a page is easy: just read the page. Flash chips do this quite well, and quickly; in terms of performance, they offer the potential to greatly exceed the random read performance of modern disk drives, which are slow due to mechanical seek and rotation costs.

Writing a page is trickier; the entire block must first be erased (taking care to first move any data we care about to another location), and then the desired page programmed. Not only is this expensive, but frequent repetitions of this program/erase cycle can lead to the biggest reliability problem flash chips have: **wear out**. When designing a storage system with flash, the performance and reliability of writing is a central focus. We'll soon learn more about how modern SSDs attack these issues, delivering excellent performance and reliability despite these limitations.

Device	Read (μs)	Program (μs)	Erase (μs)
SLC	25	200-300	1500-2000
MLC	50	600-900	~3000
TLC	~75	~900-1350	~4500

Figure 44.2: **Raw Flash Performance Characteristics**

44.4 Flash Performance And Reliability

Because we're interested in building a storage device out of raw flash chips, it is worthwhile to understand their basic performance characteristics. Figure 44.2 presents a rough summary of some numbers found in the popular press [V12]. Therein, the author presents the basic operation latency of reads, programs, and erases across SLC, MLC, and TLC flash, which store 1, 2, and 3 bits of information per cell, respectively.

As we can see from the table, read latencies are quite good, taking just 10s of microseconds to complete. Program latency is higher and more variable, as low as 200 microseconds for SLC, but higher as you pack more bits into each cell; to get good write performance, you will have to make use of multiple flash chips in parallel. Finally, erases are quite expensive, taking a few milliseconds typically. Dealing with this cost is central to modern flash storage design.

Let's now consider reliability of flash chips. Unlike mechanical disks, which can fail for a wide variety of reasons (including the gruesome and quite physical **head crash**, where the drive head actually makes contact with the recording surface), flash chips are pure silicon and in that sense have fewer reliability issues to worry about. The primary concern is **wear out**; when a flash block is erased and programmed, it slowly accrues a little bit of extra charge. Over time, as that extra charge builds up, it becomes increasingly difficult to differentiate between a 0 and a 1. At the point where it becomes impossible, the block becomes unusable.

The typical lifetime of a block is currently not well known. Manufacturers rate MLC-based blocks as having a 10,000 P/E (Program/Erase) cycle lifetime; that is, each block can be erased and programmed 10,000 times before failing. SLC-based chips, because they store only a single bit per transistor, are rated with a longer lifetime, usually 100,000 P/E cycles. However, recent research has shown that lifetimes are much longer than expected [BD10].

One other reliability problem within flash chips is known as **disturbance**. When accessing a particular page within a flash, it is possible that some bits get flipped in neighboring pages; such bit flips are known as **read disturbs** or **program disturbs**, depending on whether the page is being read or programmed, respectively.

TIP: THE IMPORTANCE OF BACKWARDS COMPATIBILITY
Backwards compatibility is always a concern in layered systems. By defining a stable interface between two systems, one enables innovation on each side of the interface while ensuring continued interoperability. Such an approach has been quite successful in many domains: operating systems have relatively stable APIs for applications, disks provide the same block-based interface to file systems, and each layer in the IP networking stack provides a fixed unchanging interface to the layer above.

Not surprisingly, there can be a downside to such rigidity, as interfaces defined in one generation may not be appropriate in the next. In some cases, it may be useful to think about redesigning the entire system entirely. An excellent example is found in the Sun ZFS file system [B07]; by reconsidering the interaction of file systems and RAID, the creators of ZFS envisioned (and then realized) a more effective integrated whole.

44.5 From Raw Flash to Flash-Based SSDs

Given our basic understanding of flash chips, we now face our next task: how to turn a basic set of flash chips into something that looks like a typical storage device. The standard storage interface is a simple block-based one, where blocks (sectors) of size 512 bytes (or larger) can be read or written, given a block address. The task of the flash-based SSD is to provide that standard block interface atop the raw flash chips inside it.

Internally, an SSD consists of some number of flash chips (for persistent storage). An SSD also contains some amount of volatile (i.e., non-persistent) memory (e.g., SRAM); such memory is useful for caching and buffering of data as well as for mapping tables, which we'll learn about below. Finally, an SSD contains control logic to orchestrate device operation. See Agrawal et. al for details [A+08]; a simplified block diagram is seen in Figure 44.3 (page 7).

One of the essential functions of this control logic is to satisfy client reads and writes, turning them into internal flash operations as need be. The **flash translation layer**, or **FTL**, provides exactly this functionality. The FTL takes read and write requests on *logical blocks* (that comprise the device interface) and turns them into low-level read, erase, and program commands on the underlying *physical blocks* and *physical pages* (that comprise the actual flash device). The FTL should accomplish this task with the goal of delivering excellent performance and high reliability.

Excellent performance, as we'll see, can be realized through a combination of techniques. One key will be to utilize multiple flash chips in **parallel**; although we won't discuss this technique much further, suffice it to say that all modern SSDs use multiple chips internally to obtain higher performance. Another performance goal will be to reduce **write amplification**, which is defined as the total write traffic (in bytes) issued to the flash chips by the FTL divided by the total write traffic (in bytes) is-

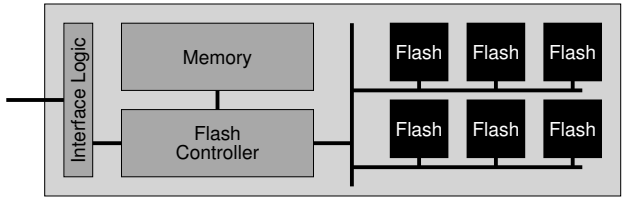


Figure 44.3: **A Flash-based SSD: Logical Diagram**

sued by the client to the SSD. As we'll see below, naive approaches to FTL construction will lead to high write amplification and low performance.

High reliability will be achieved through the combination of a few different approaches. One main concern, as discussed above, is **wear out**. If a single block is erased and programmed too often, it will become unusable; as a result, the FTL should try to spread writes across the blocks of the flash as evenly as possible, ensuring that all of the blocks of the device wear out at roughly the same time; doing so is called **wear leveling** and is an essential part of any modern FTL.

Another reliability concern is program disturbance. To minimize such disturbance, FTLs will commonly program pages within an erased block *in order*, from low page to high page. This sequential-programming approach minimizes disturbance and is widely utilized.

44.6 FTL Organization: A Bad Approach

The simplest organization of an FTL would be something we call **direct mapped**. In this approach, a read to logical page N is mapped directly to a read of physical page N. A write to logical page N is more complicated; the FTL first has to read in the entire block that page N is contained within; it then has to erase the block; finally, the FTL programs the old pages as well as the new one.

As you can probably guess, the direct-mapped FTL has many problems, both in terms of performance as well as reliability. The performance problems come on each write: the device has to read in the entire block (costly), erase it (quite costly), and then program it (costly). The end result is severe write amplification (proportional to the number of pages in a block) and as a result, terrible write performance, even slower than typical hard drives with their mechanical seeks and rotational delays.

Even worse is the reliability of this approach. If file system metadata or user file data is repeatedly overwritten, the same block is erased and programmed, over and over, rapidly wearing it out and potentially losing data. The direct mapped approach simply gives too much control over wear out to the client workload; if the workload does not spread write load evenly across its logical blocks, the underlying physical blocks containing popular data will quickly wear out. For both reliability and performance reasons, a direct-mapped FTL is a bad idea.

44.7 A Log-Structured FTL

For these reasons, most FTLs today are **log structured**, an idea useful in both storage devices (as we'll see now) and file systems above them (e.g., in **log-structured file systems**). Upon a write to logical block N, the device appends the write to the next free spot in the currently-being-written-to block; we call this style of writing **logging**. To allow for subsequent reads of block N, the device keeps a **mapping table** (in its memory, and persistent, in some form, on the device); this table stores the physical address of each logical block in the system.

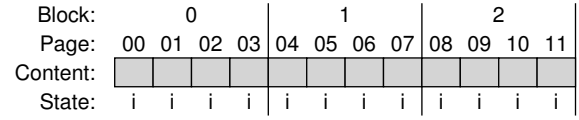
Let's go through an example to make sure we understand how the basic log-based approach works. To the client, the device looks like a typical disk, in which it can read and write 512-byte sectors (or groups of sectors). For simplicity, assume that the client is reading or writing 4-KB sized chunks. Let us further assume that the SSD contains some large number of 16-KB sized blocks, each divided into four 4-KB pages; these parameters are unrealistic (flash blocks usually consist of more pages) but will serve our didactic purposes quite well.

Assume the client issues the following sequence of operations:

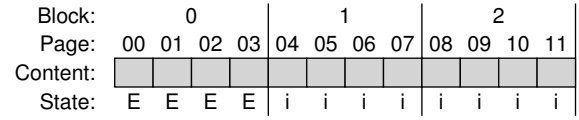
- Write(100) with contents `a1`
- Write(101) with contents `a2`
- Write(2000) with contents `b1`
- Write(2001) with contents `b2`

These **logical block addresses** (e.g., 100) are used by the client of the SSD (e.g., a file system) to remember where information is located.

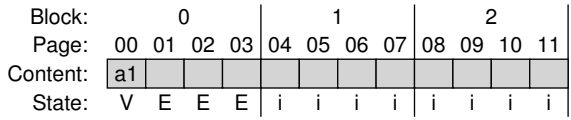
Internally, the device must transform these block writes into the erase and program operations supported by the raw hardware, and somehow record, for each logical block address, which **physical page** of the SSD stores its data. Assume that all blocks of the SSD are currently not valid, and must be erased before any page can be programmed. Here we show the initial state of our SSD, with all pages marked `INVALID` (`i`):



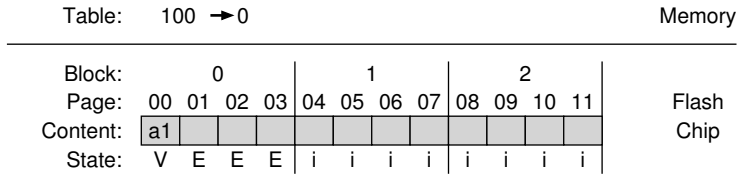
When the first write is received by the SSD (to logical block 100), the FTL decides to write it to physical block 0, which contains four physical pages: 0, 1, 2, and 3. Because the block is not erased, we cannot write to it yet; the device must first issue an erase command to block 0. Doing so leads to the following state:



Block 0 is now ready to be programmed. Most SSDs will write pages in order (i.e., low to high), reducing reliability problems related to **program disturbance**. The SSD then directs the write of logical block 100 into physical page 0:

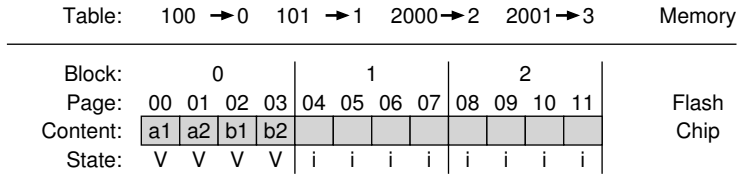


But what if the client wants to *read* logical block 100? How can it find where it is? The SSD must transform a read issued to logical block 100 into a read of physical page 0. To accommodate such functionality, when the FTL writes logical block 100 to physical page 0, it records this fact in an **in-memory mapping table**. We will track the state of this mapping table in the diagrams as well:



Now you can see what happens when the client writes to the SSD. The SSD finds a location for the write, usually just picking the next free page; it then programs that page with the block's contents, and records the logical-to-physical mapping in its mapping table. Subsequent reads simply use the table to **translate** the logical block address presented by the client into the physical page number required to read the data.

Let's now examine the rest of the writes in our example write stream: 101, 2000, and 2001. After writing these blocks, the state of the device is:



The log-based approach by its nature improves performance (erases only being required once in a while, and the costly read-modify-write of the direct-mapped approach avoided altogether), and greatly enhances reliability. The FTL can now spread writes across all pages, performing what is called **wear leveling** and increasing the lifetime of the device; we'll discuss wear leveling further below.

ASIDE: FTL MAPPING INFORMATION PERSISTENCE

You might be wondering: what happens if the device loses power? Does the in-memory mapping table disappear? Clearly, such information cannot truly be lost, because otherwise the device would not function as a persistent storage device. An SSD must have some means of recovering mapping information.

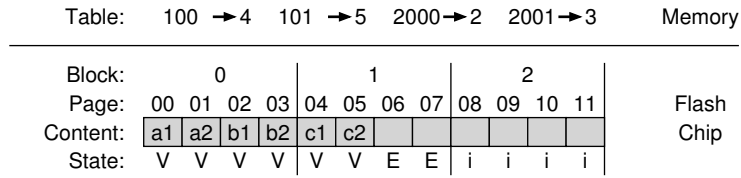
The simplest thing to do is to record some mapping information with each page, in what is called an **out-of-band** (**OOB**) area. When the device loses power and is restarted, it must reconstruct its mapping table by scanning the OOB areas and reconstructing the mapping table in memory. This basic approach has its problems; scanning a large SSD to find all necessary mapping information is slow. To overcome this limitation, some higher-end devices use more complex **logging** and **checkpointing** techniques to speed up recovery; learn more about logging by reading chapters on crash consistency and log-structured file systems [AD14a].

Unfortunately, this basic approach to log structuring has some downsides. The first is that overwrites of logical blocks lead to something we call **garbage**, i.e., old versions of data around the drive and taking up space. The device has to periodically perform **garbage collection** (**GC**) to find said blocks and free space for future writes; excessive garbage collection drives up write amplification and lowers performance. The second is high cost of in-memory mapping tables; the larger the device, the more memory such tables need. We now discuss each in turn.

44.8 Garbage Collection

The first cost of any log-structured approach such as this one is that garbage is created, and therefore **garbage collection** (i.e., dead-block reclamation) must be performed. Let's use our continued example to make sense of this. Recall that logical blocks 100, 101, 2000, and 2001 have been written to the device.

Now, let's assume that blocks 100 and 101 are written to again, with contents `c1` and `c2`. The writes are written to the next free pages (in this case, physical pages 4 and 5), and the mapping table is updated accordingly. Note that the device must have first erased block 1 to make such programming possible:



The problem we have now should be obvious: physical pages 0 and 1, although marked `VALID`, have **garbage** in them, i.e., the old versions of blocks 100 and 101. Because of the log-structured nature of the device, overwrites create garbage blocks, which the device must reclaim to provide free space for new writes to take place.

The process of finding garbage blocks (also called **dead blocks**) and reclaiming them for future use is called **garbage collection**, and it is an important component of any modern SSD. The basic process is simple: find a block that contains one or more garbage pages, read in the live (non-garbage) pages from that block, write out those live pages to the log, and (finally) reclaim the entire block for use in writing.

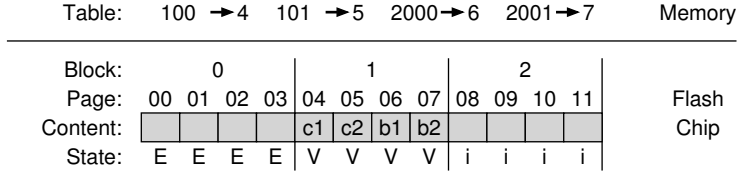
Let's now illustrate with an example. The device decides it wants to reclaim any dead pages within block 0 above. Block 0 has two dead blocks (pages 0 and 1) and two live blocks (pages 2 and 3, which contain blocks 2000 and 2001, respectively). To do so, the device will:

- Read live data (pages 2 and 3) from block 0
- Write live data to end of the log
- Erase block 0 (freeing it for later usage)

For the garbage collector to function, there must be enough information within each block to enable the SSD to determine whether each page is live or dead. One natural way to achieve this end is to store, at some location within each block, information about which logical blocks are stored within each page. The device can then use the mapping table to determine whether each page within the block holds live data or not.

From our example above (before the garbage collection has taken place), block 0 held logical blocks 100, 101, 2000, 2001. By checking the mapping table (which, before garbage collection, contained `100->4, 101->5, 2000->2, 2001->3`), the device can readily determine whether each of the pages within the SSD block holds live information. For example, pages 2 and 3 are clearly still pointed to by the map; pages 0 and 1 are not and therefore are candidates for garbage collection.

When this garbage collection process is complete in our example, the state of the device is:



As you can see, garbage collection can be expensive, requiring reading and rewriting of live data. The ideal candidate for reclamation is a block that consists of only dead pages; in this case, the block can immediately be erased and used for new data, without expensive data migration.

ASIDE: A NEW STORAGE API KNOWN AS TRIM

When we think of hard drives, we usually just think of the most basic interface to read and write them: read and write (there is also usually some kind of **cache flush** command, ensuring that writes have actually been persisted, but sometimes we omit that for simplicity). With log-structured SSDs, and indeed, any device that keeps a flexible and changing mapping of logical-to-physical blocks, a new interface is useful, known as the **trim** operation.

The trim operation takes an address (and possibly a length) and simply informs the device that the block(s) specified by the address (and length) have been deleted; the device thus no longer has to track any information about the given address range. For a standard hard drive, trim isn't particularly useful, because the drive has a static mapping of block addresses to specific platter, track, and sector(s). For a log-structured SSD, however, it is highly useful to know that a block is no longer needed, as the SSD can then remove this information from the FTL and later reclaim the physical space during garbage collection.

Although we sometimes think of interface and implementation as separate entities, in this case, we see that the implementation shapes the interface. With complex mappings, knowledge of which blocks are no longer needed makes for a more effective implementation.

To reduce GC costs, some SSDs **overprovision** the device [A+08]; by adding extra flash capacity, cleaning can be delayed and pushed to the **background**, perhaps done at a time when the device is less busy. Adding more capacity also increases internal bandwidth, which can be used for cleaning and thus not harm perceived bandwidth to the client. Many modern drives overprovision in this manner, one key to achieving excellent overall performance.

44.9 Mapping Table Size

The second cost of log-structuring is the potential for extremely large mapping tables, with one entry for each 4-KB page of the device. With a large 1-TB SSD, for example, a single 4-byte entry per 4-KB page results in 1 GB of memory needed by the device, just for these mappings! Thus, this **page-level** FTL scheme is impractical.

Block-Based Mapping

One approach to reduce the costs of mapping is to only keep a pointer per *block* of the device, instead of per page, reducing the amount of mapping information by a factor of $\frac{Size_{block}}{Size_{page}}$. This **block-level** FTL is akin to having

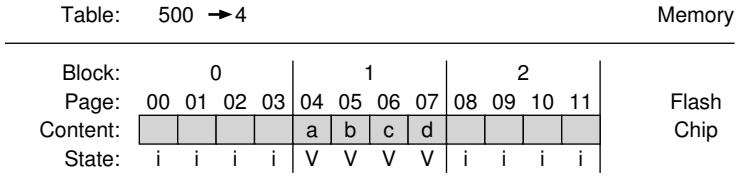
bigger page sizes in a virtual memory system; in that case, you use fewer bits for the VPN and have a larger offset in each virtual address.

Unfortunately, using a block-based mapping inside a log-based FTL does not work very well for performance reasons. The biggest problem arises when a "small write" occurs (i.e., one that is less than the size of a physical block). In this case, the FTL must read a large amount of live data from the old block and copy it into a new one (along with the data from the small write). This data copying increases write amplification greatly and thus decreases performance.

To make this issue more clear, let's look at an example. Assume the client previously wrote out logical blocks 2000, 2001, 2002, and 2003 (with contents, `a, b, c, d`), and that they are located within physical block 1 at physical pages 4, 5, 6, and 7. With per-page mappings, the translation table would have to record four mappings for these logical blocks: 2000→4, 2001→5, 2002→6, 2003→7.

If, instead, we use block-level mapping, the FTL only needs to record a single address translation for all of this data. The address mapping, however, is slightly different than our previous examples. Specifically, we think of the logical address space of the device as being chopped into chunks that are the size of the physical blocks within the flash. Thus, the logical block address consists of two portions: a chunk number and an offset. Because we are assuming four logical blocks fit within each physical block, the offset portion of the logical addresses requires 2 bits; the remaining (most significant) bits form the chunk number.

Logical blocks 2000, 2001, 2002, and 2003 all have the same chunk number (500), and have different offsets (0, 1, 2, and 3, respectively). Thus, with a block-level mapping, the FTL records that chunk 500 maps to block 1 (starting at physical page 4), as shown in this diagram:

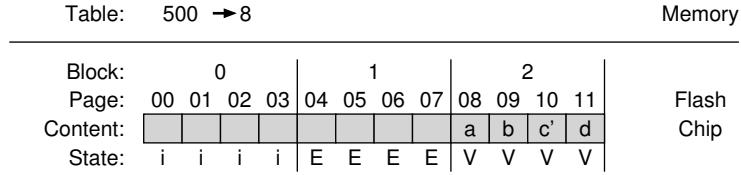


In a block-based FTL, reading is easy. First, the FTL extracts the chunk number from the logical block address presented by the client, by taking the topmost bits out of the address. Then, the FTL looks up the chunk-number to physical-page mapping in the table. Finally, the FTL computes the address of the desired flash page by *adding* the offset from the logical address to the physical address of the block.

For example, if the client issues a read to logical address 2002, the device extracts the logical chunk number (500), looks up the translation in the mapping table (finding 4), and adds the offset from the logical address (2) to the translation (4). The resulting physical-page address (6) is

where the data is located; the FTL can then issue the read to that physical address and obtain the desired data (c).

But what if the client writes to logical block 2002 (with contents c'? In this case, the FTL must read in 2000, 2001, and 2003, and then write out all four logical blocks in a new location, updating the mapping table accordingly. Block 1 (where the data used to reside) can then be erased and reused, as shown here.



As you can see from this example, while block level mappings greatly reduce the amount of memory needed for translations, they cause significant performance problems when writes are smaller than the physical block size of the device; as real physical blocks can be 256KB or larger, such writes are likely to happen quite often. Thus, a better solution is needed. Can you sense that this is the part of the chapter where we tell you what that solution is? Better yet, can you figure it out yourself, before reading on?

Hybrid Mapping

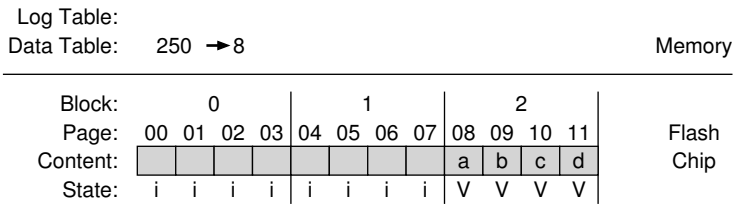
To enable flexible writing but also reduce mapping costs, many modern FTLs employ a **hybrid mapping** technique. With this approach, the FTL keeps a few blocks erased and directs all writes to them; these are called **log blocks**. Because the FTL wants to be able to write any page to any location within the log block without all the copying required by a pure block-based mapping, it keeps *per-page* mappings for these log blocks.

The FTL thus logically has two types of mapping table in its memory: a small set of per-page mappings in what we'll call the *log table*, and a larger set of per-block mappings in the *data table*. When looking for a particular logical block, the FTL will first consult the log table; if the logical block's location is not found there, the FTL will then consult the data table to find its location and then access the requested data.

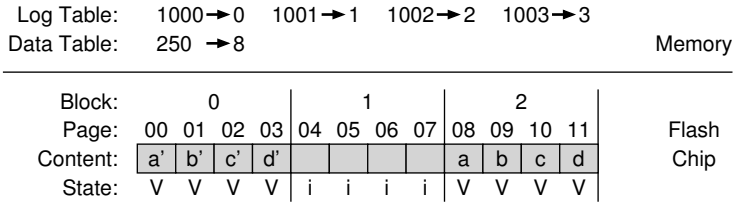
The key to the hybrid mapping strategy is keeping the number of log blocks small. To keep the number of log blocks small, the FTL has to periodically examine log blocks (which have a pointer per page) and *switch* them into blocks that can be pointed to by only a single block pointer. This switch is accomplished by one of three main techniques, based on the contents of the block [KK+02].

For example, let's say the FTL had previously written out logical pages 1000, 1001, 1002, and 1003, and placed them in physical block 2 (physical

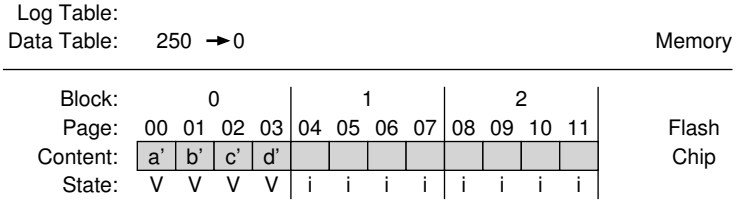
pages 8, 9, 10, 11); assume the contents of the writes to 1000, 1001, 1002, and 1003 are `a`, `b`, `c`, and `d`, respectively.



Now assume that the client overwrites each of these blocks (with data `a'`, `b'`, `c'`, and `d'`), in the exact same order, in one of the currently available log blocks, say physical block 0 (physical pages 0, 1, 2, and 3). In this case, the FTL will have the following state:

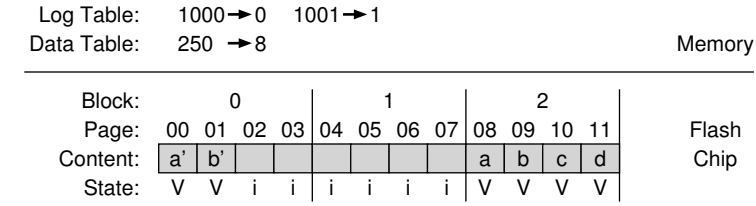


Because these blocks have been written exactly in the same manner as before, the FTL can perform what is known as a **switch merge**. In this case, the log block (0) now becomes the storage location for blocks 0, 1, 2, and 3, and is pointed to by a single block pointer; the old block (2) is now erased and used as a log block. In this best case, all the per-page pointers required replaced by a single block pointer.



This switch merge is the best case for a hybrid FTL. Unfortunately, sometimes the FTL is not so lucky. Imagine the case where we have the same initial conditions (logical blocks 1000 ... 1003 stored in physical block 2) but then the client overwrites logical blocks 1000 and 1001.

What do you think happens in this case? Why is it more challenging to handle? *(think before looking at the result on the next page)*



To reunite the other pages of this physical block, and thus be able to refer to them by only a single block pointer, the FTL performs what is called a **partial merge**. In this operation, logical blocks 1002 and 1003 are read from physical block 2, and then appended to the log. The resulting state of the SSD is the same as the switch merge above; however, in this case, the FTL had to perform extra I/O to achieve its goals, thus increasing write amplification.

The final case encountered by the FTL known as a **full merge**, and requires even more work. In this case, the FTL must pull together pages from many other blocks to perform cleaning. For example, imagine that logical blocks 0, 4, 8, and 12 are written to log block A. To switch this log block into a block-mapped page, the FTL must first create a data block containing logical blocks 0, 1, 2, and 3, and thus the FTL must read 1, 2, and 3 from elsewhere and then write out 0, 1, 2, and 3 together. Next, the merge must do the same for logical block 4, finding 5, 6, and 7 and reconciling them into a single physical block. The same must be done for logical blocks 8 and 12, and then (finally), the log block A can be freed. Frequent full merges, as is not surprising, can seriously harm performance and thus should be avoided when at all possible [GY+09].

Page Mapping Plus Caching

Given the complexity of the hybrid approach above, others have suggested simpler ways to reduce the memory load of page-mapped FTLs. Probably the simplest is just to cache only the active parts of the FTL in memory, thus reducing the amount of memory needed [GY+09].

This approach can work well. For example, if a given workload only accesses a small set of pages, the translations of those pages will be stored in the in-memory FTL, and performance will be excellent without high memory cost. Of course, the approach can also perform poorly. If memory cannot contain the **working set** of necessary translations, each access will minimally require an extra flash read to first bring in the missing mapping before being able to access the data itself. Even worse, to make room for the new mapping, the FTL might have to **evict** an old mapping, and if that mapping is **dirty** (i.e., not yet written to the flash persistently), an extra write will also be incurred. However, in many cases, the workload will display locality, and this caching approach will both reduce memory overheads and keep performance high.

44.10 Wear Leveling

Finally, a related background activity that modern FTLs must implement is **wear leveling**, as introduced above. The basic idea is simple: because multiple erase/program cycles will wear out a flash block, the FTL should try its best to spread that work across all the blocks of the device evenly. In this manner, all blocks will wear out at roughly the same time, instead of a few "popular" blocks quickly becoming unusable.

The basic log-structuring approach does a good initial job of spreading out write load, and garbage collection helps as well. However, sometimes a block will be filled with long-lived data that does not get over-written; in this case, garbage collection will never reclaim the block, and thus it does not receive its fair share of the write load.

To remedy this problem, the FTL must periodically read all the live data out of such blocks and re-write it elsewhere, thus making the block available for writing again. This process of wear leveling increases the write amplification of the SSD, and thus decreases performance as extra I/O is required to ensure that all blocks wear at roughly the same rate. Many different algorithms exist in the literature [A+08, M+14]; read more if you are interested.

44.11 SSD Performance And Cost

Before closing, let's examine the performance and cost of modern SSDs, to better understand how they will likely be used in persistent storage systems. In both cases, we'll compare to classic hard-disk drives (HDDs), and highlight the biggest differences between the two.

Performance

Unlike hard disk drives, flash-based SSDs have no mechanical components, and in fact are in many ways more similar to DRAM, in that they are "random access" devices. The biggest difference in performance, as compared to disk drives, is realized when performing random reads and writes; while a typical disk drive can only perform a few hundred random I/Os per second, SSDs can do much better. Here, we use some data from modern SSDs to see just how much better SSDs perform; we're particularly interested in how well the FTLs hide the performance issues of the raw chips.

Table 44.4 shows some performance data for three different SSDs and one top-of-the-line hard drive; the data was taken from a few different online sources [S13, T15]. The left two columns show random I/O performance, and the right two columns sequential; the first three rows show data for three different SSDs (from Samsung, Seagate, and Intel), and the last row shows performance for a **hard disk drive** (or **HDD**), in this case a Seagate high-end drive.

We can learn a few interesting facts from the table. First, and most dramatic, is the difference in random I/O performance between the SSDs

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

Figure 44.4: **SSDs And Hard Drives: Performance Comparison**

and the lone hard drive. While the SSDs obtain tens or even hundreds of MB/s in random I/Os, this "high performance" hard drive has a peak of just a couple MB/s (in fact, we rounded up to get to 2 MB/s). Second, you can see that in terms of sequential performance, there is much less of a difference; while the SSDs perform better, a hard drive is still a good choice if sequential performance is all you need. Third, you can see that SSD random read performance is not as good as SSD random write performance. The reason for such unexpectedly good random-write performance is due to the log-structured design of many SSDs, which transforms random writes into sequential ones and improves performance. Finally, because SSDs exhibit some performance difference between sequential and random I/Os, many of the techniques in chapters about how to build file systems for hard drives are still applicable to SSDs [AD14b]; although the magnitude of difference between sequential and random I/Os is smaller, there is enough of a gap to carefully consider how to design file systems to reduce random I/Os.

Cost

As we saw above, the performance of SSDs greatly outstrips modern hard drives, even when performing sequential I/O. So why haven't SSDs completely replaced hard drives as the storage medium of choice? The answer is simple: cost, or more specifically, cost per unit of capacity. Currently [A15], an SSD costs something like $150 for a 250-GB drive; such an SSD costs 60 cents per GB. A typical hard drive costs roughly $50 for 1-TB of storage, which means it costs 5 cents per GB. There is still more than a 10× difference in cost between these two storage media.

These performance and cost differences dictate how large-scale storage systems are built. If performance is the main concern, SSDs are a terrific choice, particularly if random read performance is important. If, on the other hand, you are assembling a large data center and wish to store massive amounts of information, the large cost difference will drive you towards hard drives. Of course, a hybrid approach can make sense – some storage systems are being assembled with both SSDs and hard drives, using a smaller number of SSDs for more popular "hot" data and delivering high performance, while storing the rest of the "colder" (less used) data on hard drives to save on cost. As long as the price gap exists, hard drives are here to stay.

44.12 Summary

Flash-based SSDs are becoming a common presence in laptops, desktops, and servers inside the datacenters that power the world's economy. Thus, you should probably know something about them, right?

Here's the bad news: this chapter (like many in this book) is just the first step in understanding the state of the art. Some places to get some more information about the raw technology include research on actual device performance (such as that by Chen et al. [CK+09] and Grupp et al. [GC+09]), issues in FTL design (including works by Agrawal et al. [A+08], Gupta et al. [GY+09], Huang et al. [H+14], Kim et al. [KK+02], Lee et al. [L+07], and Zhang et al. [Z+12]), and even distributed systems comprised of flash (including Gordon [CG+09] and CORFU [B+12]). And, if we may say so, a really good overview of all the things you need to do to extract high performance from an SSD can be found in a paper on the "unwritten contract" [HK+17].

Don't just read academic papers; also read about recent advances in the popular press (e.g., [V12]). Therein you'll learn more practical (but still useful) information, such as Samsung's use of both TLC and SLC cells within the same SSD to maximize performance (SLC can buffer writes quickly) as well as capacity (TLC can store more bits per cell). And this is, as they say, just the tip of the iceberg. Dive in and learn more about this "iceberg" of research on your own, perhaps starting with Ma et al.'s excellent (and recent) survey [M+14]. Be careful though; icebergs can sink even the mightiest of ships [W15].

ASIDE: KEY SSD TERMS

- A **flash chip** consists of many banks, each of which is organized into **erase blocks** (sometimes just called **blocks**). Each block is further subdivided into some number of **pages**.
- Blocks are large (128KB–2MB) and contain many pages, which are relatively small (1KB–8KB).
- To read from flash, issue a read command with an address and length; this allows a client to read one or more pages.
- Writing flash is more complex. First, the client must **erase** the entire block (which deletes all information within the block). Then, the client can **program** each page exactly once, thus completing the write.
- A new **trim** operation is useful to tell the device when a particular block (or range of blocks) is no longer needed.
- Flash reliability is mostly determined by **wear out**; if a block is erased and programmed too often, it will become unusable.
- A flash-based **solid-state storage device** (**SSD**) behaves as if it were a normal block-based read/write disk; by using a **flash translation layer** (**FTL**), it transforms reads and writes from a client into reads, erases, and programs to underlying flash chips.
- Most FTLs are **log-structured**, which reduces the cost of writing by minimizing erase/program cycles. An in-memory translation layer tracks where logical writes were located within the physical medium.
- One key problem with log-structured FTLs is the cost of **garbage collection**, which leads to **write amplification**.
- Another problem is the size of the mapping table, which can become quite large. Using a **hybrid mapping** or just **caching** hot pieces of the FTL are possible remedies.
- One last problem is **wear leveling**; the FTL must occasionally migrate data from blocks that are mostly read in order to ensure said blocks also receive their share of the erase/program load.

References

[A+08] "Design Tradeoffs for SSD Performance" by N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, R. Panigrahy. USENIX '08, San Diego California, June 2008. *An excellent overview of what goes into SSD design.*

[AD14a] "Operating Systems: Three Easy Pieces" by *Chapters: Crash Consistency: FSCK and Journaling and Log-Structured File Systems*. Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *A lot more detail here about how logging can be used in file systems; some of the same ideas can be applied inside devices too as need be.*

[AD14a] "Operating Systems: Three Easy Pieces" by *Chapters: Locality and the Fast File System and File System Implementation*. Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau. *These chapters cover how to build a basic file system for a hard drive. Amazingly, some of these ideas work perfectly well on SSDs! See if you can figure out which design techniques are appropriate, and which are less needed.*

[A15] "Amazon Pricing Study" by Remzi Arpaci-Dusseau. February, 2015. *This is not an actual paper, but rather one of the authors going to Amazon and looking at current prices of hard drives and SSDs. You too can repeat this study, and see what the costs are today. Do it!*

[B+12] "CORFU: A Shared Log Design for Flash Clusters" by M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, J. D. Davis. NSDI '12, San Jose, California, April 2012. *A new way to think about designing a high-performance replicated log for clusters using Flash.*

[BD10] "Write Endurance in Flash Drives: Measurements and Analysis" by Simona Boboila, Peter Desnoyers. FAST '10, San Jose, California, February 2010. *A cool paper that reverse engineers flash-device lifetimes. Endurance sometimes far exceeds manufacturer predictions, by up to 100×.*

[B07] "ZFS: The Last Word in File Systems" by Jeff Bonwick and Bill Moore. Available here: `http://www.ostep.org/Citations/zfs_last.pdf`. *Was this the last word in file systems? No, but maybe it's close.*

[CG+09] "Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications" by Adrian M. Caulfield, Laura M. Grupp, Steven Swanson. ASPLOS '09, Washington, D.C., March 2009. *Early research on assembling flash into larger-scale clusters; definitely worth a read.*

[CK+09] "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives" by Feng Chen, David A. Koufaty, and Xiaodong Zhang. SIGMETRICS/Performance '09, Seattle, Washington, June 2009. *An excellent overview of SSD performance problems circa 2009 (though now a little dated).*

[G14] "The SSD Endurance Experiment" by Geoff Gasior. The Tech Report, September 19, 2014. Available: `http://techreport.com/review/27062`. *A nice set of simple experiments measuring performance of SSDs over time. There are many other similar studies; use google to find more.*

[GC+09] "Characterizing Flash Memory: Anomalies, Observations, and Applications" by L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf. IEEE MICRO '09, New York, New York, December 2009. *Another excellent characterization of flash performance.*

[GY+09] "DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings" by Aayush Gupta, Youngjae Kim, Bhuvan Urgaonkar. ASPLOS '09, Washington, D.C., March 2009. *This paper gives an excellent overview of different strategies for cleaning within hybrid SSDs as well as a new scheme which saves mapping table space and improves performance under many workloads.*

[HK+17] "The Unwritten Contract of Solid State Drives" by Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. EuroSys '17, Belgrade, Serbia, April 2017. *Our own paper which lays out five rules clients should follow in order to get the best performance out of modern SSDs. The rules are request scale, locality, aligned sequentiality, grouping by death time, and uniform lifetime. Read the paper for details!*

[H+14] "An Aggressive Worn-out Flash Block Management Scheme To Alleviate SSD Performance Degradation" by Ping Huang, Guanying Wu, Xubin He, Weijun Xiao. EuroSys '14, 2014. *Recent work showing how to really get the most out of worn-out flash blocks; neat!*

[J10] "Failure Mechanisms and Models for Semiconductor Devices" by Unknown author. Report JEP122F, November 2010. Available on the internet at this exciting so-called web site: `http://www.jedec.org/sites/default/files/docs/JEP122F.pdf`. *A highly detailed discussion of what is going on at the device level and how such devices fail. Only for those not faint of heart. Or physicists. Or both.*

[KK+02] "A Space-Efficient Flash Translation Layer For Compact Flash Systems" by Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho. IEEE Transactions on Consumer Electronics, Volume 48, Number 2, May 2002. *One of the earliest proposals to suggest hybrid mappings.*

[L+07] "A Log Buffer-Based Flash Translation Layer by Using Fully-Associative Sector Translation. " Sang-won Lee, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, Ha-Joo Song. ACM Transactions on Embedded Computing Systems, Volume 6, Number 3, July 2007 *A terrific paper about how to build hybrid log/block mappings.*

[M+14] "A Survey of Address Translation Technologies for Flash Memories" by Dongzhe Ma, Jianhua Feng, Guoliang Li. ACM Computing Surveys, Volume 46, Number 3, January 2014. *Probably the best recent survey of flash and related technologies.*

[S13] "The Seagate 600 and 600 Pro SSD Review" by Anand Lal Shimpi. AnandTech, May 7, 2013. Available: `http://www.anandtech.com/show/6935/seagate-600-ssd-review`. *One of many SSD performance measurements available on the internet. Haven't heard of the internet? No problem. Just go to your web browser and type "internet" into the search tool. You'll be amazed at what you can learn.*

[T15] "Performance Charts Hard Drives" by Tom's Hardware. January 2015. Available here: `http://www.tomshardware.com/charts/enterprise-hdd-charts`. *Yet another site with performance data, this time focusing on hard drives.*

[V12] "Understanding TLC Flash" by Kristian Vatto. AnandTech, September, 2012. Available: `http://www.anandtech.com/show/5067/understanding-tlc-nand`. *A short description about TLC flash and its characteristics.*

[W15] "List of Ships Sunk by Icebergs" by Many authors. Available at this location on the "web": `http://en.wikipedia.org/wiki/List_of_ships_sunk_by_icebergs`. *Yes, there is a wikipedia page about ships sunk by icebergs. It is a really boring page and basically everyone knows the only ship the iceberg-sinking-mafia cares about is the Titanic.*

[Z+12] "De-indirection for Flash-based SSDs with Nameless Writes" by Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. FAST '13, San Jose, California, February 2013. *Our research on a new idea to reduce mapping table space; the key is to re-use the pointers in the file system above to store locations of blocks, instead of adding another level of indirection.*

Homework (Simulation)

This section introduces `ssd.py`, a simple SSD simulator you can use to understand better how SSDs work. Read the README for details on how to run the simulator. It is a long README, so boil a cup of tea (caffeinated likely necessary), put on your reading glasses, let the cat curl up on your lap[1], and get to work.

Questions

1. The homework will mostly focus on the log-structured SSD, which is simulated with the "-T log" flag. We'll use the other types of SSDs for comparison. First, run with flags `-T log -s 1 -n 10 -q`. Can you figure out which operations took place? Use `-c` to check your answers (or just use `-C` instead of `-q -c`). Use different values of `-s` to generate different random workloads.
2. Now just show the commands and see if you can figure out the intermediate states of the Flash. Run with flags `-T log -s 2 -n 10 -C` to show each command. Now, determine the state of the Flash between each command; use `-F` to show the states and see if you were right. Use different random seeds to test your burgeoning expertise.
3. Let's make this problem ever so slightly more interesting by adding the `-r 20` flag. What differences does this cause in the commands? Use `-c` again to check your answers.
4. Performance is determined by the number of erases, programs, and reads (we assume here that trims are free). Run the same workload again as above, but without showing any intermediate states (e.g., `-T log -s 1 -n 10`). Can you estimate how long this workload will take to complete? (default erase time is 1000 microseconds, program time is 40, and read time is 10) Use the `-S` flag to check your answer. You can also change the erase, program, and read times with the `-E, -W, -R` flags.
5. Now, compare performance of the log-structured approach and the (very bad) direct approach (`-T direct` instead of `-T log`). First, estimate how you think the direct approach will perform, then check your answer with the `-S` flag. In general, how much better will the log-structured approach perform than the direct one?
6. Let us next explore the behavior of the garbage collector. To do so, we have to set the high (`-G`) and low (`-g`) watermarks appropriately. First, let's observe what happens when you run a larger workload to the log-structured SSD but without any garbage collection. To do this, run with flags `-T log -n 1000` (the high wa-

[1]Now you might complain, "But I'm a dog person!" To this, we say, too bad! Get a cat, put it on your lap, and do the homework! How else will you learn, if you can't even follow the most basic of instructions?

termark default is 10, so the GC won't run in this configuration). What do you think will happen? Use `-C` and perhaps `-F` to see.

7. To turn on the garbage collector, use lower values. The high watermark (`-G N`) tells the system to start collecting once `N` blocks have been used; the low watermark (`-g M`) tells the system to stop collecting once there are only `M` blocks in use. What watermark values do you think will make for a working system? Use `-C` and `-F` to show the commands and intermediate device states and see.
8. One other useful flag is `-J`, which shows what the collector is doing when it runs. Run with flags `-T log -n 1000 -C -J` to see both the commands and the GC behavior. What do you notice about the GC? The final effect of GC, of course, is performance. Use `-S` to look at final statistics; how many extra reads and writes occur due to garbage collection? Compare this to the ideal SSD (`-T ideal`); how much extra reading, writing, and erasing is there due to the nature of Flash? Compare it also to the `direct` approach; in what way (erases, reads, programs) is the log-structured approach superior?
9. One last aspect to explore is **workload skew**. Adding skew to the workload changes writes such that more writes occur to some smaller fraction of the logical block space. For example, running with `-K 80/20` makes 80% of the writes go to 20% of the blocks. Pick some different skews and perform many randomly-chosen operations (e.g., `-n 1000`), using first `-T direct` to understand the skew, and then `-T log` to see the impact on a log-structured device. What do you expect will happen? One other small skew control to explore is `-k 100`; by adding this flag to a skewed workload, the first 100 writes are not skewed. The idea is to first create a lot of data, but then only update some of it. What impact might that have upon a garbage collector?