xv6

a simple, Unix-like teaching operating system

Russ Cox
Frans Kaashoek
Robert Morris

xv6-book@pdos.csail.mit.edu

Draft as of September 7, 2011

Contents

0 Operating system interfaces *7*

1 The first process *17*

2 Traps, interrupts, and drivers *31*

3 Locking *43*

4 Scheduling *51*

5 File system *63*

A PC hardware *79*

B The boot loader *83*

Index *89*

Foreword and acknowledgements

This is a draft text intended for a class on operating systems. It explains the main concepts of operating systems by studying an example kernel, named xv6. xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented in ANSI C for an x86-based multiprocessor.

The text should be read along with the source code for xv6. This approach is inspired by John Lions's Commentary on UNIX 6th Edition (Peer to Peer Communications; ISBN: 1-57398-013-7; 1st edition (June 14, 2000)). See http://pdos.csail.mit.edu/6.828 for pointers to on-line resources for v6 and xv6.

We have used this text in 6.828, the operating system class at MIT. We thank the faculty, TAs, and students of 6.828 who all directly or indirectly contributed to xv6. In particular, we would like to thank Austin Clements and Nickolai Zeldovich.

Chapter 0

Operating system interfaces

The job of an operating system is to share a computer among multiple programs and to provide a more useful set of services than the hardware alone supports. The operating system manages the low-level hardware, so that, for example, a word processor need not concern itself with which video card is being used. It also multiplexes the hardware, allowing many programs to share the computer and run (or appear to run) at the same time. Finally, operating systems provide controlled ways for programs to interact, so that they can share data or work together.

An operating system provides services to user programs through some interface. Designing a good interface turns out to be difficult. On the one hand, we would like the interface to be simple and narrow because that makes it easier to get the implementation right. On the other hand, we may be tempted to offer many sophisticated features to applications. The trick in resolving this tension is to design interfaces that rely on a few mechanisms that can be combined to provide much generality.

This book uses a single operating system as a concrete example to illustrate operating system concepts. That operating system, xv6, provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie's Unix operating system, as well as mimicking Unix's internal design. Unix provides a narrow interface whose mechanisms combine well, offering a surprising degree of generality. This interface has been so successful that modern operating systems—BSD, Linux, Mac OS X, Solaris, and even, to a lesser extent, Microsoft Windows—have Unix-like interfaces. Understanding xv6 is a good start toward understanding any of these systems and many others.

As shown in Figure 0-1, xv6 takes the traditional form of a `kernel`, a special program that provides services to running programs. Each running program, called a `process`, has memory containing instructions, data, and a stack. The instructions implement the program's computation. The data are the variables on which the computation acts. The stack organizes the program's procedure calls.

When a process needs to invoke a kernel service, it invokes a procedure call in the operating system interface. Such procedures are call `system calls`. The system call enters the kernel; the kernel performs the service and returns. Thus a process alternates between executing in `user space` and `kernel space`.

The kernel uses the CPU's hardware protection mechanisms to ensure that each process executing in user space can access only its own memory. The kernel executes with the hardware privileges required to implement these protections; user programs execute without those privileges. When a user program invokes a system call, the hardware raises the privilege level and starts executing a pre-arranged function in the kernel.

The collection of system calls that a kernel provides is the interface that user programs see. The xv6 kernel provides a subset of the services and system calls that Unix kernels traditionally offer. The calls are:

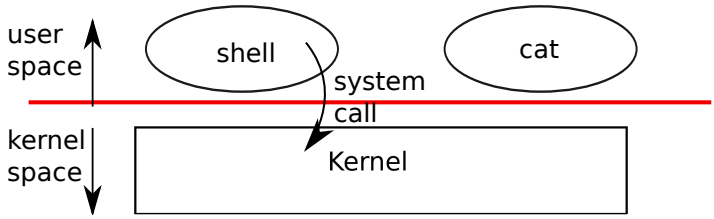

Figure 0-1. A kernel with 2 user processes.

System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n time units
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; flags indicate read/write
read(fd, buf, n)	Read n byes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

The rest of this chapter outlines xv6's services—processes, memory, file descriptors, pipes, and file system—and illustrates them with code snippets and discussions of how the shell uses them. The shell's use of system calls illustrates how carefully they have been designed.

The shell is an ordinary program that reads commands from the user and executes them, and is the primary user interface to traditional Unix-like systems. The fact that the shell is a user program, not part of the kernel, illustrates the power of the system call interface: there is nothing special about the shell. It also means that the shell is easy to replace, and modern Unix systems have a variety of shells to choose from, each with its own syntax and semantics. The xv6 shell is a simple implementation of the essence of the Unix Bourne shell. Its implementation can be found at line (7650).

Code: Processes and memory

An xv6 process consists of user-space memory (instructions, data, and stack) and

per-process state private to the kernel. Xv6 provides time-sharing: it transparently switches the available CPUs among the set of processes waiting to execute. When a process is not executing, xv6 saves its CPU registers, restoring them when it next runs the process. Each process can be uniquely identified by a positive integer called its process identifier, or `pid`.

A process may create a new process using the `fork` system call. `Fork` creates a new process, called the `child process`, with exactly the same memory contents as the calling process, called the `parent process`. `Fork` returns in both the parent and the child. In the parent, `fork` returns the child's pid; in the child, it returns zero. For example, consider the following program fragment:

```
int pid;

pid = fork();
if(pid > 0){
  printf("parent: child=%d\n", pid);
  pid = wait();
  printf("child %d is done\n", pid);
} else if(pid == 0){
  printf("child: exiting\n");
  exit();
} else {
  printf("fork error\n");
}
```

The `exit` system call causes the calling process to stop executing and to release resources such as memory and open files. The `wait` system call returns the pid of an exited child of the current process; if none of the caller's children has exited, `wait` waits for one to do so. In the example, the output lines

```
parent: child=1234
child: exiting
```

might come out in either order, depending on whether the parent or child gets to its `printf` call first. After those two, the child exits, and then the parent's `wait` returns, causing the parent to print

```
parent: child 1234 is done
```

Note that the parent and child were executing with different memory and different registers: changing a variable in one does not affect the other.

The `exec` system call replaces the calling process's memory with a new memory image loaded from a file stored in the file system. The file must have a particular format, which specifies which part of the file holds instructions, which part is data, at which instruction to start, etc.. The format xv6 uses is called the ELF format, which Chapter 1 discusses in more detail. When `exec` succeeds, it does not return to the calling program; instead, the instructions loaded from the file start executing at the entry point declared in the ELF header. `Exec` takes two arguments: the name of the file containing the executable and an array of string arguments. For example:

```
char *argv[3];

argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo", argv);
printf("exec error\n");
```

This fragment replaces the calling program with an instance of the program `/bin/echo` running with the argument list `echo hello`. Most programs ignore the first argument, which is conventionally the name of the program.

The xv6 shell uses the above calls to run programs on behalf of users. The main structure of the shell is simple; see `main` on line (7801). The main loop reads the input on the command line using `getcmd`. Then it calls `fork`, which creates another running shell program. The parent shell calls `wait`, while the child process runs the command. For example, if the user had typed "`echo hello`" at the prompt, `runcmd` would have been called with "`echo hello`" as the argument. `runcmd` (7706) runs the actual command. For "`echo hello`", it would call `exec` on line (7726). If `exec` succeeds then the child will execute instructions from `echo` instead of `runcmd`. At some point `echo` will call `exit`, which will cause the parent to return from `wait` in `main` (7801). You might wonder why `fork` and `exec` are not combined in a single call; we will see later that separate calls for creating a process and loading a program is a clever design.

Xv6 allocates most user-space memory implicitly: `fork` allocates the memory required for the child's copy of the parent's memory, and `exec` allocates enough memory to hold the executable file. A process that needs more memory at run-time (perhaps for `malloc`) can call `sbrk(n)` to grow its data memory by `n` bytes; `sbrk` returns the location of the new memory.

Xv6 does not provide a notion of users or of protecting one user from another; in Unix terms, all xv6 processes run as root.

Code: File descriptors

A `file descriptor` is a small integer representing a kernel-managed object that a process may read from or write to. A process may obtain a file descriptor by opening a file, directory, or device, or by creating a pipe, or by duplicating an existing descriptor. Internally, the xv6 kernel uses the file descriptor as an index into a per-process table, so that every process has a private space of file descriptors starting at zero. By convention, a process reads from file descriptor 0 (standard input), writes output to file descriptor 1 (standard output), and writes error messages to file descriptor 2 (standard error). As we will see, the shell exploits the convention to implement I/O redirection and pipelines. The shell ensures that it always has three file descriptors open (7807), which are by default file descriptors for the console.

The `read` and `write` system calls read bytes from and write bytes to open files named by file descriptors. The call `read(fd, buf, n)` reads at most `n` bytes from the file descriptor `fd`, copies them into `buf`, and returns the number of bytes read. Every file descriptor has an offset associated with it. `Read` reads data from the current file offset and then advances that offset by the number of bytes read: a subsequent `read`

will return the bytes following the ones returned by the first `read`. When there are no more bytes to read, `read` returns zero to signal the end of the file.

The call `write(fd, buf, n)` writes `n` bytes from `buf` to the file descriptor `fd` and returns the number of bytes written. Fewer than `n` bytes are written only when an error occurs. Like `read`, `write` writes data at the current file offset and then advances that offset by the number of bytes written: each `write` picks up where the previous one left off.

The following program fragment (which forms the essence of `cat`) copies data from its standard input to its standard output. If an error occurs, it writes a message to the standard error.

```
char buf[512];
int n;

for(;;){
  n = read(0, buf, sizeof buf);
  if(n == 0)
    break;
  if(n < 0){
    fprintf(2, "read error\n");
    exit();
  }
  if(write(1, buf, n) != n){
    fprintf(2, "write error\n");
    exit();
  }
}
```

The important thing to note in the code fragment is that `cat` doesn't know whether it is reading from a file, console, or a pipe. Similarly `cat` doesn't know whether it is printing to a console, a file, or whatever. The use of file descriptors and the convention that file descriptor 0 is input and file descriptor 1 is output allows a simple implementation of `cat`.

The `close` system call releases a file descriptor, making it free for reuse by a future `open`, `pipe`, or `dup` system call (see below). A newly allocated file descriptor is always the lowest-numbered unused descriptor of the current process.

File descriptors and `fork` interact to make I/O redirection easy to implement. `Fork` copies then parent's file descriptor table along with its memory, so that the child starts with exactly the same open files as the parent. The system call `exec` replaces the calling process's memory but preserves its file table. This behavior allows the shell to implement I/O redirection by forking, reopening chosen file descriptors, and then execing the new program. Here is a simplified version of the code a shell runs for the command `cat <input.txt`:

```
char *argv[2];

argv[0] = "cat";
argv[1] = 0;
if(fork() == 0) {
  close(0);
  open("input.txt", O_RDONLY);
```

```
    exec("cat", argv);
  }
```

After the child closes file descriptor 0, open is guaranteed to use that file descriptor for the newly opened `input.txt`: 0 will be the smallest available file descriptor. `Cat` then executes with file descriptor 0 (standard input) referring to `input.txt`.

The code for I/O redirection in the xv6 shell works exactly in this way (7730). Recall that at this point in the code the shell has already forked the child shell and that `runcmd` will call `exec` to load the new program. Now it should be clear why it is a good idea that `fork` and `exec` are separate calls. This separation allows the shell to fix up the child process before the child runs the intended program.

Although `fork` copies the file descriptor table, each underlying file offset is shared between parent and child. Consider this example:

```
if(fork() == 0) {
  write(1, "hello ", 6);
  exit();
} else {
  wait();
  write(1, "world\n", 6);
}
```

At the end of this fragment, the file attached to file descriptor 1 will contain the data `hello world`. The `write` in the parent (which, thanks to `wait`, runs only after the child is done) picks up where the child's `write` left off. This behavior helps produce useful results from sequences of shell commands, like `(echo hello; echo world) >output.txt`.

The `dup` system call duplicates an existing file descriptor, returning a new one that refers to the same underlying I/O object. Both file descriptors share an offset, just as the file descriptors duplicated by `fork` do. This is another way to write `hello world` into a file:

```
fd = dup(1);
write(1, "hello ", 6);
write(fd, "world\n", 6);
```

Two file descriptors share an offset if they were derived from the same original file descriptor by a sequence of `fork` and `dup` calls. Otherwise file descriptors do not share offsets, even if they resulted from `open` calls for the same file. `Dup` allows shells to implement commands like this: `ls existing-file non-existing-file > tmp1 2>&1`. The `2>&1` tells the shell to give the command a file descriptor 2 that is a duplicate of descriptor 1. Both the name of the existing file and the error message for the non-existing file will show up in the file `tmp1`. The xv6 shell doesn't support I/O redirection for the error file descriptor, but now you can implement it.

File descriptors are a powerful abstraction, because they hide the details of what they are connected to: a process writing to file descriptor 1 may be writing to a file, to a device like the console, or to a pipe.

Code: Pipes

A pipe is a small kernel buffer exposed to processes as a pair of file descriptors, one for reading and one for writing. Writing data to one end of the pipe makes that data available for reading from the other end of the pipe. Pipes provide a way for processes to communicate.

The following example code runs the program wc with standard input connected to the read end of a pipe.

```
int p[2];
char *argv[2];

argv[0] = "wc";
argv[1] = 0;

pipe(p);
if(fork() == 0) {
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  exec("/bin/wc", argv);
} else {
  write(p[1], "hello world\n", 12);
  close(p[0]);
  close(p[1]);
}
```

The program calls pipe, which creates a new pipe and records the read and write file descriptors in the array p. After fork, both parent and child have file descriptors referring to the pipe. The child dups the read end onto file descriptor 0, closes the file descriptors in p, and execs wc. When wc reads from its standard input, it reads from the pipe. The parent writes to the write end of the pipe and then closes both of its file descriptors.

If no data is available, a read on a pipe waits for either data to be written or all file descriptors referring to the write end to be closed; in the latter case, read will return 0, just as if the end of a data file had been reached. The fact that read blocks until it is impossible for new data to arrive is one reason that it's important for the child to close the write end of the pipe before executing wc above: if one of wc's file descriptors referred to the write end of the pipe, wc would never see end-of-file.

The xv6 shell implements pipes in a manner similar to the above code (7750). The child process creates a pipe to connect the left end of the pipe with the right end of the pipe. Then it calls runcmd for the left part of the pipe and runcmd for the right end of the pipe, and waits for the left and the right end to finish, by calling wait twice. The right end of the pipe may be a command that itself includes a pipe (e.g., a | b | c), which itself forks two new child processes (one for b and one for c). Thus, the shell may create a tree of processes. The leaves of this tree are commands and the interior nodes are processes that wait until the left and right children complete. In principle, you could have the interior nodes run the left end of a pipe, but doing so correctly will complicate the implementation.

Pipes may seem no more powerful than temporary files: the pipeline

```
echo hello world | wc
```

could also be implemented without pipes as

```
echo hello world >/tmp/xyz; wc </tmp/xyz
```

There are at least three key differences between pipes and temporary files. First, pipes automatically clean themselves up; with the file redirection, a shell would have to be careful to remove `/tmp/xyz` when done. Second, pipes can pass arbitrarily long streams of data, while file redirection requires enough free space on disk to store all the data. Third, pipes allow for synchronization: two processes can use a pair of pipes to send messages back and forth to each other, with each `read` blocking its calling process until the other process has sent data with `write`.

Code: File system

Xv6 provides data files, which are uninterpreted byte arrays, and directories, which contain named references to other data files and directories. Xv6 implements directories as a special kind of file. The directories are arranged into a tree, starting at a special directory called the `root`. A `path` like `/a/b/c` refers to the file or directory named `c` inside the directory named `b` inside the directory named `a` in the root directory `/`. Paths that don't begin with `/` are evaluated relative to the calling process's `current directory`, which can be changed with the `chdir` system call. Both these code fragments open the same file:

```
chdir("/a");
chdir("b");
open("c", O_RDONLY);

open("/a/b/c", O_RDONLY);
```

The first changes the process's current directory to `/a/b`; the second neither refers to nor modifies the process's current directory.

There are multiple system calls to create a new file or directory: `mkdir` creates a new directory, `open` with the `O_CREATE` flag creates a new data file, and `mknod` creates a new device file. This example illustrates all three:

```
mkdir("/dir");
fd = open("/dir/file", O_CREATE|O_WRONLY);
close(fd);
mknod("/console", 1, 1);
```

`Mknod` creates a file in the file system, but the file has no contents. Instead, the file's metadata marks it as a device file and records the major and minor device numbers (the two arguments to `mknod`), which uniquely identify a kernel device. When a process later opens the file, the kernel diverts `read` and `write` system calls to the kernel device implementation instead of passing them to the file system.

`fstat` retrieves information about the object a file descriptor refers to. It fills in a `struct stat`, defined in `stat.h` as:

```
#define T_DIR  1   // Directory
#define T_FILE 2   // File
#define T_DEV  3   // Special device

struct stat {
  short type;  // Type of file
  int dev;     // Device number
  uint ino;    // Inode number on device
  short nlink; // Number of links to file
  uint size;   // Size of file in bytes
};
```

A file's name is distinct from the file itself; the same underlying file, called an `inode`, can have multiple names, called `links`. The `link` system call creates another file system name referring to the same inode as an existing file. This fragment creates a new file named both `a` and `b`.

```
open("a", O_CREATE|O_WRONLY);
link("a", "b");
```

Reading from or writing to `a` is the same as reading from or writing to `b`. Each inode is identified by a unique *inode number*. After the code sequence above, it is possible to determine that `a` and `b` refer to the same underlying contents by inspecting the result of `fstat`: both will return the same inode number (`ino`), and the `nlink` count will be set to 2.

The `unlink` system call removes a name from the file system. The file's inode and the disk space holding its content are only freed when the file's link count is zero and no file descriptors refer to it. Thus adding

```
unlink("a");
```

to the last code sequence leaves the inode and file content accessible as `b`. Furthermore,

```
fd = open("/tmp/xyz", O_CREATE|O_RDWR);
unlink("/tmp/xyz");
```

is an idiomatic way to create a temporary inode that will be cleaned up when the process closes `fd` or exits.

Xv6 commands for file system operations are implemented as user-level programs such as `mkdir`, `ln`, `rm`, etc. This design allows anyone to extend the shell with new user commands. In hind-sight this plan seems obvious, but other systems designed at the time of Unix often built such commands into the shell (and built the shell into the kernel).

One exception is `cd`, which built into the shell (7816). The reason is that cd must change the current working directory of the shell itself. If `cd` were run as a regular command, then the shell would fork a child process, the child process would run `cd`, change the *child*'s working directory, and then return to the parent. The parent's (i.e., the shell's) working directory would not change.

Real world

Unix's combination of the "standard" file descriptors, pipes, and convenient shell syntax for operations on them was a major advance in writing general-purpose reusable programs. The idea sparked a whole culture of "software tools" that was responsible for much of Unix's power and popularity, and the shell was the first so-called "scripting language." The Unix system call interface persists today in systems like BSD, Linux, and Mac OS X.

Modern kernels provide many more system calls, and many more kinds of kernel services, than xv6. For the most part, modern Unix-derived operating systems have not followed the early Unix model of exposing devices as special files, like the `console` device file discussed above. The authors of Unix went on to build Plan 9, which applied the "resources are files" concept to modern facilities, representing networks, graphics, and other resources as files or file trees.

The file system as an interface has been a very powerful idea, most recently applied to network resources in the form of the World Wide Web. Even so, there are other models for operating system interfaces. Multics, a predecessor of Unix, blurred the distinction between data in memory and data on disk, producing a very different flavor of interface. The complexity of the Multics design had a direct influence on the designers of Unix, who tried to build something simpler.

This book examines how xv6 implements its Unix-like interface, but the ideas and concepts apply to more than just Unix. Any operating system must multiplex processes onto the underlying hardware, isolate processes from each other, and provide mechanisms for controlled inter-process communication. After studying xv6, you should be able to look at other, more complex operating systems and see the concepts underlying xv6 in those systems as well.

Chapter 1

The first process

This chapter explains what happens when xv6 first starts running, through the creation of the first process. One of the most interesting aspects of this is how the kernel manages memory for itself and for processes.

One purpose of processes is to isolate different programs that share the same computer, so that one buggy program cannot break others. A process provides a program with what appears to be a private memory system, or address space, which other processes cannot read or write. This chapter examines how xv6 configures the processor's paging hardware to provide processes with private address spaces, how it allocates memory to hold process code and data, and how it creates new processes.

Paging hardware

Xv6 runs on Intel 80386 or later ("x86") processors on a PC platform, and much of its low-level functionality (for example, its virtual memory implementation) is x86-specific. This book assumes the reader has done a bit of machine-level programming on some architecture, and will introduce x86-specific ideas as they come up. Appendix A briefly outlines the PC platform.

The x86 paging hardware uses a page table to translate (or "map") a `virtual address` (the address that an x86 instruction manipulates) to a `physical address` (an address that the processor chip sends to main memory).

An x86 page table is logically an array of 2^20 (1,048,576) `page table entries (PTEs)`. Each PTE contains a 20-bit physical page number (PPN) and some flags. The paging hardware translates a virtual address by using its top 20 bits to index into the page table to find a PTE, and replacing those bits with the PPN in the PTE. The paging hardware copies the low 12 bits unchanged from the virtual to the translated physical address. Thus a page table gives the operating system control over virtual-to-physical address translations at the granularity of aligned chunks of 4096 (2^12) bytes. Such a chunk is called a `page`.

As shown in Figure 1-1, the actual translation happens in two steps. A page table is stored in physical memory as a two-level tree. The root of the tree is a 4096-byte `page directory` that contains 1024 PTE-like references to `page table pages`. Each page table page is an array of 1024 32-bit PTEs. The paging hardware uses the top 10 bits of a virtual address to select a page directory entry. If the page directory entry is present, the paging hardware uses the next 10 bits of the virtual address to select a PTE from the page table page that the page directory entry refers to. If either the page directory entry or the PTE is not present, the paging hardware raises a fault. This two-level structure allows a page table to omit entire page table pages in the common case in which large ranges of virtual addresses have no mappings.

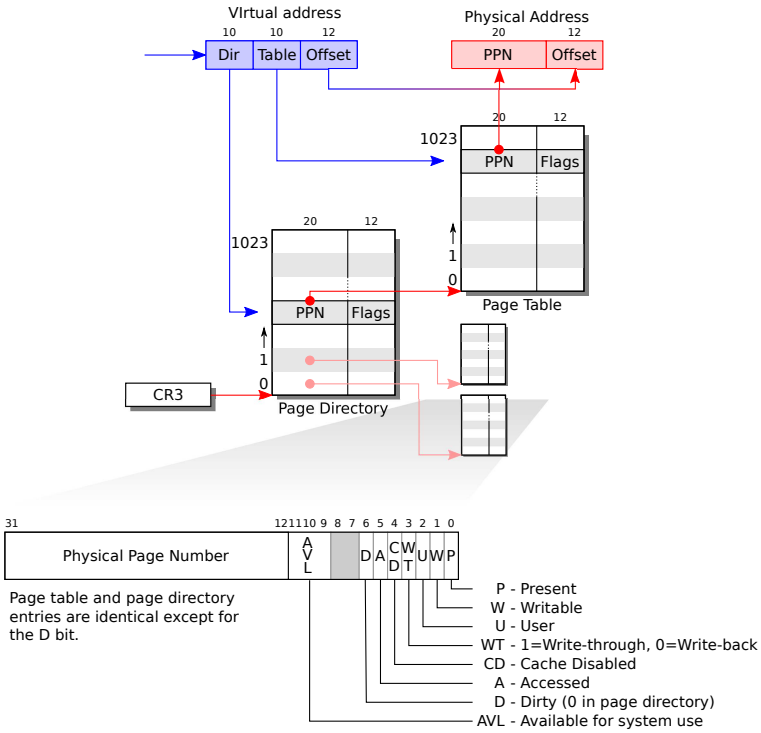

Figure 1-1. Page tables on x86.

Each PTE contains flag bits that tell the paging hardware how the associated virtual address is allowed to be used. `PTE_P` indicates whether the PTE is present: if it is not set, a reference to the page causes a fault (i.e. is not allowed). `PTE_W` controls whether instructions are allowed to issue writes to the page; if not set, only reads and instruction fetches are allowed. `PTE_U` controls whether user programs are allowed to use the page; if clear, only the kernel is allowed to use the page. Figure 1-1 shows how it all works.

A few notes about terms. Physical memory refers to storage cells in DRAM. A byte of physical memory has an address, called a physical address. A program uses virtual addresses, which the paging hardware translate to physical addresses, and then send to the DRAM hardware to read or write storage. At this level of discussion there is no such thing as virtual memory, only virtual addresses.

Address space overview

Xv6 uses the paging hardware to give each process its own view of memory, called an `address space`. Xv6 maintains a separate page table for each process that defines that process's address space. An address space includes the process's `user memory` starting at virtual address zero. Instructions usually come first, followed by global variables and a "heap" area (for malloc) that the process can expand as needed.

Each process's address space maps the kernel's instructions and data as well as the user program's memory. When a process invokes a system call, the system call exe-

cutes in the kernel mappings of the process's address space. This arrangement exists so that the kernel's system call code can directly refer to user memory. In order to leave room for user memory to grow, xv6's address spaces map the kernel at high addresses, starting at 0x80100000.

Code: entry page table

When a PC powers on, it initializes itself and then loads a `boot loader` from disk into memory and executes it. Appendix B explains the details. Xv6's boot loader loads the xv6 kernel from disk and executes it starting at `entry` (1040). The x86 paging hardware is not enabled when the kernel starts; virtual addresses map directly to physical addresses.

The boot loader loads the xv6 kernel into memory at physical address 0x100000. The reason it doesn't load the kernel at 0x80100000, where the kernel expects to find its instructions and data, is that there may not be any physical memory at such a high address on a small machine. The reason it places the kernel at 0x100000 rather than 0x0 is because the address range 0xa0000:0x100000 contains older I/O devices. To allow the rest of the kernel to run, `entry` sets up a page table that maps virtual addresses starting at 0x80000000 (called `KERNBASE` (0207)) to physical address starting at 0x0.

The entry page table is defined in main.c (1317). The array initialization sets two of the 1024 PTEs, at indices zero and 960 (`KERNBASE>>PDXSHIFT`), leaving the other PTEs zero. It causes both of these PTEs to use a `superpage`, which maps 4 megabytes of virtual address space. Entry 0 maps virtual addresses 0:0x400000 to physical addresses 0:0x400000. This mapping is required as long as `entry` is executing at low addresses, but will eventually be removed. The pages are mapped as present `PTE_P` (present), `PTE_W` (writeable), and `PTE_PS` (super page). The flags and all other page hardware related structures are defined in `mmu.h` (0200).

Entry 960 maps virtual addresses KERNBASE:KERNBASE+0x400000 to physical addresses 0:0x400000. This entry will be used by the kernel after `entry` has finished; it maps the high virtual addresses at which the kernel expects to find its instructions and data to the low physical addresses where the boot loader loaded them. This mapping restricts the kernel instructions and data to 4 Mbytes.

Returning to `entry`, the kernel first tells the paging hardware to allow super pages by setting the flag `CR_PSE` (page size extension) in the control register `%cr4`. Next it loads the physical address of `entrypgdir` into control register `%cr3`. The paging hardware must know the physical address of `entrypgdir`, because it doesn't know how to translate virtual addresses yet; it doesn't have a page table yet. The symbol `entrypgdir` refers to an address in high memory, and the macro `V2P_WO` (0220) subtracts `KERNBASE` in order to find the physical address. To enable the paging hardware, xv6 sets the flag `CR0_PG` in the control register `%cr0`. It also sets `CR0_WP`, which ensures that the kernel honors write-protect flags in PTEs.

The processor is still executing instructions at low addresses after paging is enabled, which works since `entrypgdir` maps low addresses. If xv6 had omitted entry 0 from `entrypgdir`, the computer would have crashed when trying to execute the in-

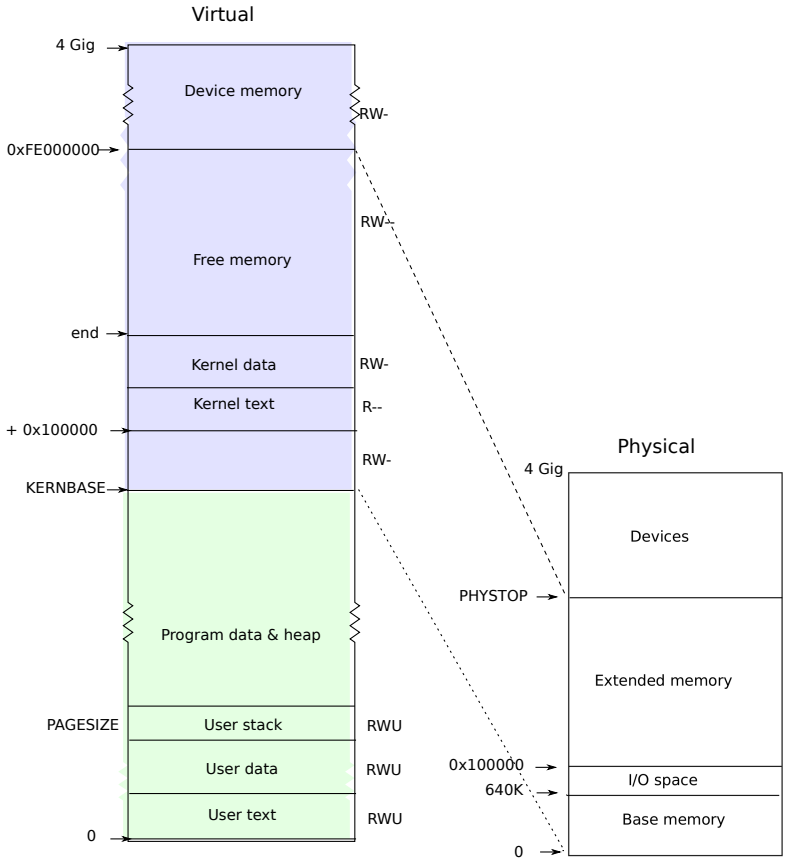

Figure 1-2. Layout of a virtual address space and the physical address space.

struction after the one that enabled paging.

Now `entry` needs to transfer to the kernel's C code, and run it in high memory. First it must make the stack pointer, `%esp`, point to a stack so that C code will work (1054). All symbols have high addresses, including `stack`, so the stack will still be valid even when the low mappings are removed. Finally `entry` jumps to `main`, which is also a high address. The indirect jump is needed because the assembler would generate a PC-relative direct jump, which would execute the low-memory version of `main`. Main cannot return, since the there's no return PC on the stack. Now the kernel is running in high addresses in the function `main` (1216).

Address space details

The page table created by `entry` has enough mappings to allow the kernel's C code to start running. However, `main` immediately changes to a new page table by calling `kvmalloc` (1753), because kernel has a more elaborate plan for page tables that describe process address spaces.

Each process has a separate page table, and xv6 tells the page table hardware to switch page tables when xv6 switches between processes. As shown in Figure 1-2, a process's user memory starts at virtual address zero and can grow up to `KERNBASE`, al-

lowing a process to address up to 2 GB of memory. When a process asks xv6 for more memory, xv6 first finds free physical pages to provide the storage, and then adds PTEs to the process's page table that point to the new physical pages. xv6 sets the PTE_U, PTE_W, and PTE_P flags in these PTEs. Most processes do not use the entire user address space; xv6 leaves PTE_P clear in unused PTEs. Different processes' page tables translate user addresses to different pages of physical memory, so that each process has private user memory.

Xv6 includes all mappings needed for the kernel to run in every process's page table; these mappings all appear above KERNBASE. It maps virtual addresses KERNBASE:KERNBASE+PHYSTOP to 0:PHYSTOP. One reason for this mapping is so that the kernel can use its own instructions and data. Another reason is that the kernel sometimes needs to be able to write a given page of physical memory, for example when creating page table pages; having every physical page appear at a predictable virtual address makes this convenient. A defect of this arrangement is that xv6 cannot make use of more than 2 GB of physical memory. Some devices that use memory-mapped I/O appear at physical addresses starting at 0xFE000000, so xv6 page tables including a direct mapping for them. Xv6 does not set the PTE_U flag in the PTEs above KERNBASE, so only the kernel can use them.

Having every process's page table contain mappings for both user memory and the entire kernel is convenient when switching from user code to kernel code during system calls and interrupts: such switches do not require page table switches. For the most part the kernel does not have its own page table; it is almost always borrowing some process's page table.

To review, xv6 ensures that each process can only use its own memory, and that a process sees its memory as having contiguous virtual addresses. xv6 implements the first by setting the PTE_U bit only on PTEs of virtual addresses that refer to the process's own memory. It implements the second using the ability of page tables to translate successive virtual addresses to whatever physical pages happen to be allocated to the process.

Code: creating an address space

main calls kvmalloc (1753) to create and switch to a page table with the mappings above KERNBASE required for the kernel to run. Most of the work happens in setupkvm (1734). It first allocates a page of memory to hold the page directory. Then it calls mappages to install the translations that the kernel needs, which are described in the kmap (1725) array. The translations include the kernel's instructions and data, physical memory up to PHYSTOP, and memory ranges which are actually I/O devices. setupkvm does not install any mappings for the user memory; this will happen later.

mappages (1679) installs mappings into a page table for a range of virtual addresses to a corresponding range of physical addresses. It does this separately for each virtual address in the range, at page intervals. For each virtual address to be mapped, mappages calls walkpgdir to find the address of the PTE for that address. It then initializes the PTE to hold the relevant physical page number, the desired permissions (PTE_W and/or PTE_U), and PTE_P to mark the PTE as valid (1692).

`walkpgdir` (1654) mimics the actions of the x86 paging hardware as it looks up the PTE for a virtual address (see Figure 1-1). `walkpgdir` uses the upper 10 bits of the virtual address to find the page directory entry (1659). If the page directory entry isn't present, then the required page table page hasn't yet been allocated; if the `alloc` argument is set, `walkpgdir` allocates it and puts its physical address in the page directory. Finally it uses the next 10 bits of the virtual address to find the address of the PTE in the page table page (1672).

Physical memory allocation

The kernel needs to allocate and free physical memory at run-time for page tables, process user memory, kernel stacks, and pipe buffers.

xv6 uses the physical memory between the end of the kernel and `PHYSTOP` for run-time allocation. It allocates and frees whole 4096-byte pages at a time. It keeps track of which pages are free by threading a linked list through the pages themselves. Allocation consists of removing a page from the linked list; freeing consists of adding the freed page to the list.

There is a bootstrap problem: all of physical memory must be mapped in order for the allocator to initialize the free list, but creating a page table with those mappings involves allocating page-table pages. xv6 solves this problem by using a separate page allocator during entry, which allocates memory just after the end of the kernel's data segment. This allocator does not support freeing and is limited by the 4 MB mapping in the `entrypgdir`, but that is sufficient to allocate the first kernel page table.

Code: Physical memory allocator

The allocator's data structure is a *free list* of physical memory pages that are available for allocation. Each free page's list element is a `struct run` (2711). Where does the allocator get the memory to hold that data structure? It store each free page's `run` structure in the free page itself, since there's nothing else stored there. The free list is protected by a spin lock (2711-2713). The list and the lock are wrapped in a struct to make clear that the lock protects the fields in the struct. For now, ignore the lock and the calls to `acquire` and `release`; Chapter 3 will examine locking in detail.

The function `main` calls `kinit` to initialize the allocator (2740). `kinit` ought to determine how much physical memory is available, but this turns out to be difficult on the x86. Instead it assumes that the machine has 240 megabytes (`PHYSTOP`) of physical memory, and uses all the memory between the end of the kernel and `PHYSTOP` as the initial pool of free memory. `kinit` calls `kfree` with the address of each page of memory between `end` and `PHYSTOP`. This causes `kfree` to add those pages to the allocator's free list. The allocator starts with no memory; these calls to `kfree` give it some to manage.

The allocator refers to physical pages by their virtual addresses as mapped in high memory, not by their physical addresses, which is why `kinit` uses `p2v(PHYSTOP)` to translate `PHYSTOP` (a physical address) to a virtual address. The allocator sometimes treats addresses as integers in order to perform arithmetic on them (e.g., traversing all

pages in kinit), and sometimes uses addresses as pointers to read and write memory (e.g., manipulating the run structure stored in each page); this dual use of addresses is the main reason that the allocator code is full of C type casts. The other reason is that freeing and allocation inherently change the type of the memory.

The function kfree (2756) begins by setting every byte in the memory being freed to the value 1. This will cause code that uses memory after freeing it (uses "dangling references") to read garbage instead of the old valid contents; hopefully that will cause such code to break faster. Then kfree casts v to a pointer to struct run, records the old start of the free list in r->next, and sets the free list equal to r. kalloc removes and returns the first element in the free list.

When creating the first kernel page table, setupkvm and walkpgdir use enter_alloc (2725) instead of kalloc. This memory allocator moves the end of the kernel by 1 page. enter_alloc uses the symbol end, which the linker causes to have an address that is just beyond the end of the kernel's data segment. A PTE can only refer to a physical address that is aligned on a 4096-byte boundary (is a multiple of 4096), so enter_alloc uses PGROUNDUP to ensure that it allocates only aligned physical addresses. Memory allocated with enter_alloc is never freed.

Code: Process creation

This section describes how xv6 creates the first process. The xv6 kernel maintains many pieces of state for each process, which it gathers into a struct proc (2053). A process's most important pieces of kernel state are its page table and the physical memory it refers to, its kernel stack, and its run state. We'll use the notation p->xxx to refer to elements of the proc structure.

You should view the kernel state of a process as a thread of execution (or thread for short) that executes in the kernel on behalf of a process. A thread executes a computation but can be stopped and then resumed. For example, when a process makes a system call, the CPU switches from executing the process to executing the process's kernel thread. The process's kernel thread executes the implementation of the system call (e.g., reads a file), and then returns back to the process. xv6 executes a system call as a thread so that a system can wait (or "block") in the kernel to wait for I/O, and resume where it left off when the I/O has finished. Much of the state of a kernel thread (local variables, functional call return addresses) is stored on the kernel thread's stack, p->kstack. Each process's kernel stack is separate from its user stack, since the user stack may not be valid. So, you can view a process has having two threads of execution: one user thread and one kernel thread.

p->state indicates whether the process is allocated, ready to run, running, waiting for I/O, or exiting.

p->pgdir holds the process's page table, an array of PTEs. xv6 causes the paging hardware to use a process's p->pgdir when executing that process. A process's page table also serves as the record of the addresses of the physical pages allocated to store the process's memory.

The story of the creation of the first process starts when main (1237) calls userinit (2202), whose first action is to call allocproc. The job of allocproc (2155) is

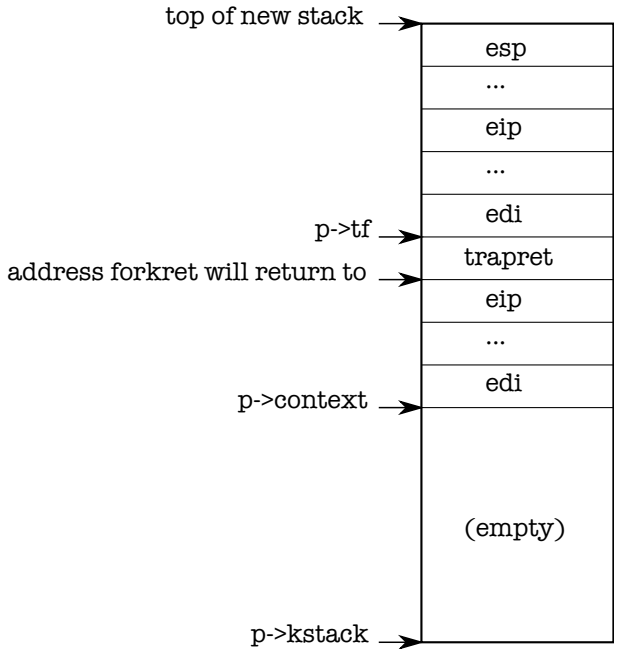

Figure 1-3. Set up of a new kernel stack.

to allocate a slot (a `struct proc`) in the process table and to initialize the parts of the process's state required for its kernel thread to execute. `Allocproc` is called for all new processes, while `userinit` is called only for the very first process. `Allocproc` scans the table for a process with state `UNUSED` (2161-2163). When it finds an unused process, `allocproc` sets the state to `EMBRYO` to mark it as used and gives the process a unique `pid` (2151-2169). Next, it tries to allocate a kernel stack for the process's kernel thread. If the memory allocation fails, `allocproc` changes the state back to `UNUSED` and returns zero to signal failure.

Now `allocproc` must set up the new process's kernel stack. Ordinarily processes are created only by `fork`, so a new process starts life copied from its parent. The result of `fork` is a child process that has identical memory contents to its parent. `allocproc` sets up the child to start life running its kernel thread, with a specially prepared kernel stack and set of kernel registers that cause it to "return" to user space at the same place (the return from the `fork` system call) as the parent. The layout of the prepared kernel stack will be as shown in Figure 1-3. `allocproc` does part of this work by setting up return program counter values that will cause the new process's kernel thread to first execute in `forkret` and then in `trapret` (2186-2191). The kernel thread will start executing with register contents copied from `p->context`. Thus setting `p->context->eip` to `forkret` will cause the kernel thread to execute at the start of `forkret` (2483). This function will return to whatever address is at the bottom of the stack. The context switch code (2658) sets the stack pointer to point just beyond the end of `p->context`. `allocproc` places `p->context` on the stack, and puts a pointer to `trapret` just above it; that is where `forkret` will return. `trapret` restores user registers from values stored at the top of the kernel stack and jumps into the process

(2927). This setup is the same for ordinary `fork` and for creating the first process, though in the latter case the process will start executing at location zero rather than at a return from `fork`.

As we will see in Chapter 2, the way that control transfers from user software to the kernel is via an interrupt mechanism, which is used by system calls, interrupts, and exceptions. Whenever control transfers into the kernel while a process is running, the hardware and xv6 trap entry code save user registers on the top of the process's kernel stack. `userinit` writes values at the top of the new stack that look just like those that would be there if the process had entered the kernel via an interrupt (2214-2220), so that the ordinary code for returning from the kernel back to the process's user code will work. These values are a `struct trapframe` which stores the user registers. Now the new process's kernel stack is completely prepared as shown in Figure 1-3.

The first process is going to execute a small program (`initcode.S`; (7500)). The process needs physical memory in which to store this program, the program needs to be copied to that memory, and the process needs a page table that refers to that memory.

`userinit` calls `setupkvm` (1734) to create a page table for the process with (at first) mappings only for memory that the kernel uses. This function is the same one that the kernel used to setup its page table.

The initial contents of the first process's memory are the compiled form of `initcode.S`; as part of the kernel build process, the linker embeds that binary in the kernel and defines two special symbols `_binary_initcode_start` and `_binary_initcode_size` telling the location and size of the binary. `Userinit` copies that binary into the new process's memory by calling `inituvm`, which allocates one page of physical memory, maps virtual address zero to that memory, and copies the binary to that page (1786).

Then `userinit` sets up the trap frame with the initial user mode state: the `%cs` register contains a segment selector for the `SEG_UCODE` segment running at privilege level `DPL_USER` (i.e., user mode not kernel mode), and similarly `%ds`, `%es`, and `%ss` use `SEG_UDATA` with privilege `DPL_USER`. The `%eflags FL_IF` is set to allow hardware interrupts; we will reexamine this in Chapter 2.

The stack pointer `%esp` is the process's largest valid virtual address, `p->sz`. The instruction pointer is the entry point for the initcode, address 0.

The function `userinit` sets `p->name` to `initcode` mainly for debugging. Setting `p->cwd` sets the process's current working directory; we will examine `namei` in detail in Chapter 5.

Once the process is initialized, `userinit` marks it available for scheduling by setting `p->state` to `RUNNABLE`.

Code: Running a process

Now that the first process's state is prepared, it is time to run it. After `main` calls `userinit`, `mpmain` calls `scheduler` to start running processes (1267). `Scheduler` (2408) looks for a process with `p->state` set to `RUNNABLE`, and there's only one it can find: `initproc`. It sets the per-cpu variable `proc` to the process it found and calls `switchu-`

vm to tell the hardware to start using the target process's page table (1764). Changing page tables while executing in the kernel works because setupkvm causes all processes' page tables to have identical mappings for kernel code and data. switchuvm also creates a new task state segment SEG_TSS that instructs the hardware to handle an interrupt by returning to kernel mode with %ss and %esp set to SEG_KDATA <<3 and (uint)proc->kstack+KSTACKSIZE, the top of this process's kernel stack. We will reexamine the task state segment in Chapter 2.

scheduler now sets p->state to RUNNING and calls swtch (2658) to perform a context switch to the target process's kernel thread. swtch saves the current registers and loads the saved registers of the target kernel thread (proc->context) into the x86 hardware registers, including the stack pointer and instruction pointer. The current context is not a process but rather a special per-cpu scheduler context, so scheduler tells swtch to save the current hardware registers in per-cpu storage (cpu->scheduler) rather than in any process's kernel thread context. We'll examine switch in more detail in Chapter 4. The final ret instruction (2677) pops a new %eip from the stack, finishing the context switch. Now the processor is running the kernel thread of process p.

Allocproc set initproc's p->context->eip to forkret, so the ret starts executing forkret. Forkret releases the ptable.lock (see Chapter 3). On the first invocation (that is this one), forkret (2483) runs initialization functions that cannot be run from main because they must be run in the context of a regular process with its own kernel stack. Then, forkret returns. Allocproc arranged that the top word on the stack after p->context is popped off would be trapret, so now trapret begins executing, with %esp set to p->tf. Trapret (2927) uses pop instructions to walk up the trap frame just as swtch did with the kernel context: popal restores the general registers, then the popl instructions restore %gs, %fs, %es, and %ds. The addl skips over the two fields trapno and errcode. Finally, the iret instructions pops %cs, %eip, and %flags off the stack. The contents of the trap frame have been transferred to the CPU state, so the processor continues at the %eip specified in the trap frame. For initproc, that means virtual address zero, the first instruction of initcode.S.

At this point, %eip holds zero and %esp holds 4096. These are virtual addresses in the process's address space. The processor's paging hardware translates them into physical addresses. allocuvm set up the PTE for the page at virtual address zero to point to the physical memory allocated for this process, and marked that PTE with PTE_U so that the process can use it. No other PTEs in the process's page table have the PTE_U bit set. The fact that userinit (2214) set up the low bits of %cs to run the process's user code at CPL=3 means that the user code can only use PTE entries with PTE_U set, and cannot modify sensitive hardware registers such as %cr3. So the process is constrained to using only its own memory.

Initcode.S (7508) begins by pushing three values on the stack—$argv, $init, and $0—and then sets %eax to SYS_exec and executes int T_SYSCALL: it is asking the kernel to run the exec system call. If all goes well, exec never returns: it starts running the program named by $init, which is a pointer to the NUL-terminated string /init (7521-7523). If the exec fails and does return, initcode loops calling the exit system call, which definitely should not return (7515-7519).

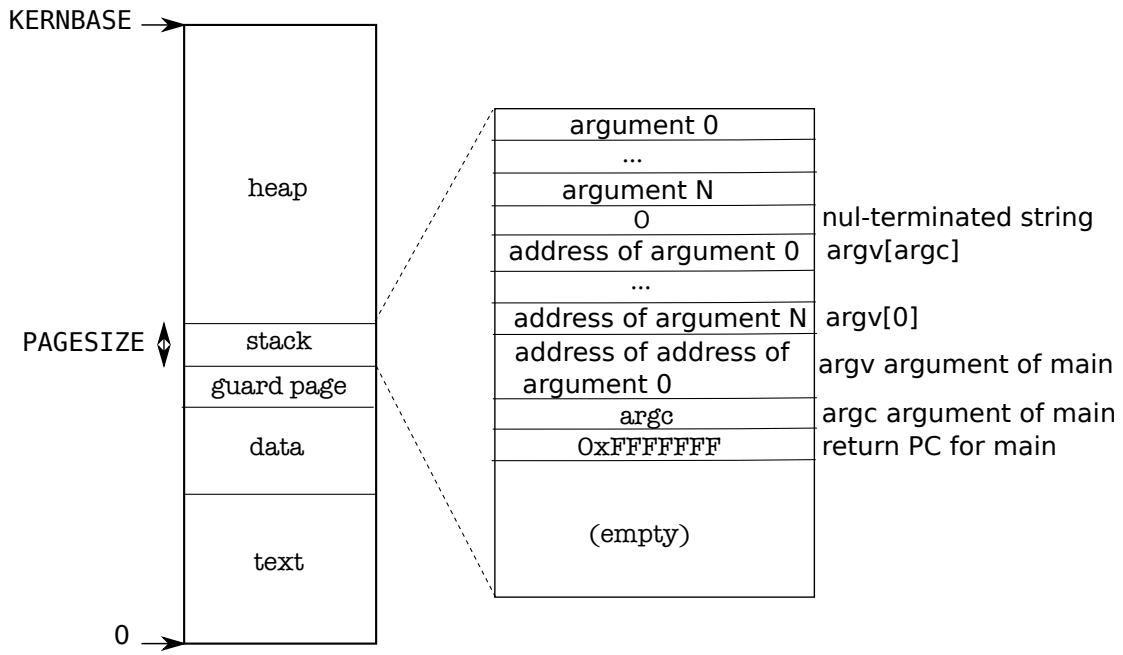

Figure 1-4. Memory layout of a user process with its initial stack.

The arguments to the `exec` system call are `$init` and `$argv`. The final zero makes this hand-written system call look like the ordinary system calls, as we will see in Chapter 2. As before, this setup avoids special-casing the first process (in this case, its first system call), and instead reuses code that xv6 must provide for standard operation.

Exec

As we saw in Chapter 0, `exec` replaces the memory and registers of the current process with a new program, but it leaves the file descriptors, process id, and parent process the same.

Figure 1-4 shows the user memory image of an executing process. The heap is above the stack so that it can expand (with `sbrk`). The stack is a single page, and is shown with the initial contents as created by exec. Strings containing the command-line arguments, as well as an array of pointers to them, are at the very top of the stack. Just under that are values that allow a program to start at `main` as if the function call `main(argc, argv)` had just started.

Code: exec

When the system call arrives (Chapter 2 will explain how that happens), `syscall` invokes `sys_exec` via the `syscalls` table (3250). `Sys_exec` (5620) parses the system call arguments (also explained in Chapter 2), and invokes `exec` (5642).

Exec (5710) opens the named binary path using namei (5720), which is explained in Chapter 5, and then reads the ELF header. Xv6 applications are described in the widely-used ELF format, defined in elf.h. An ELF binary consists of an ELF header, struct elfhdr (0955), followed by a sequence of program section headers, struct proghdr (0974). Each proghdr describes a section of the application that must be loaded into memory; xv6 programs have only one program section header, but other systems might have separate sections for instructions and data.

The first step is a quick check that the file probably contains an ELF binary. An ELF binary starts with the four-byte "magic number" 0x7F, 'E', 'L', 'F', or ELF_MAGIC (0952). If the ELF header has the right magic number, exec assumes that the binary is well-formed.

Then exec allocates a new page table with no user mappings with setupkvm (5731), allocates memory for each ELF segment with allocuvm (5743), and loads each segment into memory with loaduvm (5745). The program section header for /init looks like this:

```
# objdump -p _init

_init:     file format elf32-i386

Program Header:
    LOAD off    0x00000054 vaddr 0x00000000 paddr 0x00000000 align 2**2
         filesz 0x000008c0 memsz 0x000008cc flags rwx
```

allocuvm checks that the virtual addresses requested is below KERNBASE. loaduvm (1803) uses walkpgdir to find the physical address of the allocated memory at which to write each page of the ELF segment, and readi to read from the file.

The program section header's filesz may be less than the memsz, indicating that the gap between them should be filled with zeroes (for C global variables) rather than read from the file. For /init, filesz is 2240 bytes and memsz is 2252 bytes, and thus allocuvm allocates enough physical memory to hold 2252 bytes, but reads only 2240 bytes from the file /init.

Now exec allocates and initializes the user stack. It allocates just one stack page. It also places an inaccessible page just below the stack page, so that programs that try to use more than one page will fault. This inaccessible page also allows exec to deal with arguments that are too large; in that situation, the copyout function that exec uses to copy arguments to the stack will notice that the destination page in not accessible, and will return –1.

Exec copies the argument strings to the top of the stack one at a time, recording the pointers to them in ustack. It places a null pointer at the end of what will be the argv list passed to main. The first three entries in ustack are the fake return PC, argc, and argv pointer.

During the preparation of the new memory image, if exec detects an error like an invalid program segment, it jumps to the label bad, frees the new image, and returns –1. Exec must wait to free the old image until it is sure that the system call will succeed: if the old image is gone, the system call cannot return –1 to it. The only error cases in exec happen during the creation of the image. Once the image is complete, exec can install the new image (5789) and free the old one (5790). Finally, exec

returns 0. Success!

Now the `initcode` (7500) is done. `Exec` has replaced it with the `/init` binary, loaded out of the file system. `Init` (7610) creates a new console device file if needed and then opens it as file descriptors 0, 1, and 2. Then it loops, starting a console shell, handles orphaned zombies until the shell exits, and repeats. The system is up.

Although the system is up, we skipped over some important subsystems of xv6. The next chapter examines how xv6 configures the x86 hardware to handle the system call interrupt caused by `int $T_SYSCALL`. The rest of the book explains process management and the file system.

Real world

Most operating systems have adopted the process concept, and most processes look similar to xv6's. A real operating system would find free `proc` structures with an explicit free list in constant time instead of the linear-time search in `allocproc`; xv6 uses the linear scan (the first of many) for simplicity.

Like most operating systems, xv6 uses the paging hardware for memory protection and mapping. Most operating systems make far more sophisticated use of paging than xv6; for example, xv6 lacks demand paging from disk, copy-on-write fork, shared memory, and automatically extending stacks. The x86 also supports address translation using segmentation (see Appendix B), but xv6 uses them only for the common trick of implementing per-cpu variables such as `proc` that are at a fixed address but have different values on different CPUs (see `seginit`). Implementations of per-CPU (or per-thread) storage on non-segment architectures would dedicate a register to holding a pointer to the per-CPU data area, but the x86 has so few general registers that the extra effort required to use segmentation is worthwhile.

xv6's address space layout has the defect that it cannot make use of more than 2 GB of physical RAM. It's possible to fix this, though the best plan would be to switch to a machine with 64-bit addresses.

Xv6 should determine the actual RAM configuration, instead of assuming 240 MB. On the x86, there are at least three common algorithms: the first is to probe the physical address space looking for regions that behave like memory, preserving the values written to them; the second is to read the number of kilobytes of memory out of a known 16-bit location in the PC's non-volatile RAM; and the third is to look in BIOS memory for a memory layout table left as part of the multiprocessor tables. Reading the memory layout table is complicated.

Memory allocation was a hot topic a long time ago, the basic problems being efficient use of very limited memory and preparing for unknown future requests. See Knuth. Today people care more about speed than space-efficiency. In addition, a more elaborate kernel would likely allocate many different sizes of small blocks, rather than (as in xv6) just 4096-byte blocks; a real kernel allocator would need to handle small allocations as well as large ones.

`Exec` is the most complicated code in xv6. It involves pointer translation (in `sys_exec` too), many error cases, and must replace one running process with another. Real world operationg systems have even more complicated `exec` implementations.

They handle shell scripts (see exercise below), more complicated ELF binaries, and even multiple binary formats.

Exercises

1. Set a breakpoint at swtch. Single step with gdb's `stepi` through the ret to `forkret`, then use gdb's `finish` to proceed to `trapret`, then `stepi` until you get to `initcode` at virtual address zero.

2. Look at real operating systems to see how they size memory.

3. If xv6 had not used super pages, what would be the right declaration for `entrypgdir`?

4. Unix implementations of `exec` traditionally include special handling for shell scripts. If the file to execute begins with the text `#!`, then the first line is taken to be a program to run to interpret the file. For example, if `exec` is called to run `myprog arg1` and `myprog`'s first line is `#!/interp`, then `exec` runs `/interp` with command line `/interp myprog arg1`. Implement support for this convention in xv6.

5. `KERNBASE` limits the amount of memory a single process can use, which might be irritating on a machine with a full 4 GB of RAM. Would raising `KERNBASE` allow a process to use more memory?

Chapter 2

Traps, interrupts, and drivers

When running a process, a CPU executes the normal processor loop: read an instruction, advance the program counter, execute the instruction, repeat. But there are events on which control from a user program must transferred back to the kernel instead of executing the next instruction. These events include a device signaling that it wants attention, a user program doing something illegal (e.g., references a virtual address for which there is no PTE), or a user program asking the kernel for a service with a system call. There are three main challenges in handling these events: 1) the kernel must arrange that a processor switches from user mode to kernel mode (and back); 2) the kernel and devices must coordinate their parallel activities; and 3) the kernel must understand the interface of the devices well. Addressing these 3 challenges requires detailed understanding of hardware and careful programming, and can result in opaque kernel code. This chapter explains how xv6 addresses these three challenges.

Systems calls, exceptions, and interrupts

With a system call a user program can ask for an operating system service, as we saw at the end of the last chapter. The term `exception` refers to an illegal program action that generates an interrupt. Examples of illegal programs actions include divide by zero, attempt to access memory for a PTE that is not present, and so on. The term `interrupt` refers to a signal generated by a hardware device, indicating that it needs attention of the operating system. For example, a clock chip may generate an interrupt every 100 msec to allow the kernel to implement time sharing. As another example, when the disk has read a block from disk, it generates an interrupt to alert the operating system that the block is ready to be retrieved.

The kernel handles all interrupts, rather than processes handling them, because in most cases only the kernel has the required privilege and state. For example, in order to time-slice among processes in response the clock interrupts, the kernel must be involved, if only to force uncooperative processes to yield the processor.

In all three cases, the operating system design must arrange for the following to happen. The system must save the processor's registers for future transparent resume. The system must be set up for execution in the kernel. The system must chose a place for the kernel to start executing. The kernel must be able to retrieve information about the event, e.g., system call arguments. It must all be done securely; the system must maintain isolation of user processes and the kernel.

To achieve this goal the operating system must be aware of the details of how the hardware handles system calls, exceptions, and interrupts. In most processors these three events are handled by a single hardware mechanism. For example, on the x86, a

program invokes a system call by generating an interrupt using the `int` instruction. Similarly, exceptions generate an interrupt too. Thus, if the operating system has a plan for interrupt handling, then the operating system can handle system calls and exceptions too.

The basic plan is as follows. An interrupts stops the normal processor loop and starts executing a new sequence called an `interrupt handler`. Before starting the interrupt handler, the processor saves its registers, so that the operating system can restore them when it returns from the interrupt. A challenge in the transition to and from the interrupt handler is that the processor should switch from user mode to kernel mode, and back.

A word on terminology: Although the official x86 term is interrupt, x86 refers to all of these as `traps`, largely because it was the term used by the PDP11/40 and therefore is the conventional Unix term. This chapter uses the terms trap and interrupt interchangeably, but it is important to remember that traps are caused by the current process running on a processor (e.g., the process makes a system call and as a result generates a trap), and interrupts are caused by devices and may not be related to the currently running process. For example, a disk may generate an interrupt when it is done retrieving a block for one process, but at the time of the interrupt some other process may be running. This property of interrupts makes thinking about interrupts more difficult than thinking about traps, because interrupts happen concurrently with other activities. Both rely, however, on the same hardware mechanism to transfer control between user and kernel mode securely, which we will discuss next.

X86 protection

The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege). In practice, most operating systems use only 2 levels: 0 and 3, which are then called `kernel mode` and `user mode`, respectively. The current privilege level with which the x86 executes instructions is stored in `%cs` register, in the field CPL.

On the x86, interrupt handlers are defined in the interrupt descriptor table (IDT). The IDT has 256 entries, each giving the `%cs` and `%eip` to be used when handling the corresponding interrupt.

To make a system call on the x86, a program invokes the `int` *n* instruction, where *n* specifies the index into the IDT. The `int` instruction performs the following steps:

- Fetch the *n*'th descriptor from the IDT, where *n* is the argument of `int`.
- Check that CPL in `%cs` is <= DPL, where DPL is the privilege level in the descriptor.
- Save `%esp` and `%ss` in a CPU-internal registers, but only if the target segment selector's PL < CPL.
- Load `%ss` and `%esp` from a task segment descriptor.
- Push `%ss`.

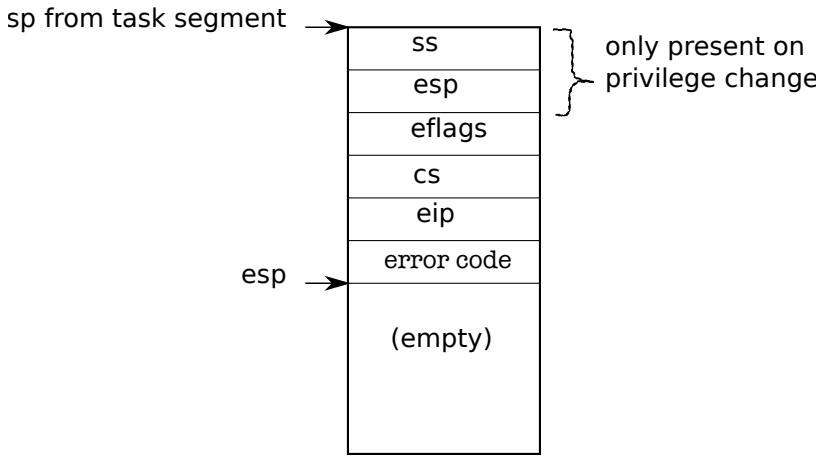

Figure 2-1. Kernel stack after an int instruction.

- Push `%esp`.
- Push `%eflags`.
- Push `%cs`.
- Push `%eip`.
- Clear some bits of `%eflags`.
- Set `%cs` and `%eip` to the values in the descriptor.

The `int` instruction is a complex instruction, and one might wonder whether all these actions are necessary. The check CPL <= DPL allows the kernel to forbid systems for some privilege levels. For example, for a user program to execute `int` instruction succesfully, the DPL must be 3. If the user program doesn't have the appropriate privilege, then `int` instruction will result in `int` 13, which is a general protection fault. As another example, the `int` instruction cannot use the user stack to save values, because the user might not have set up an appropriate stack so that hardware uses the stack specified in the task segments, which is setup in kernel mode.

Figure 2-1 shows the stack after an `int` instruction completes and there was a privilege-level change (the privilege level in the descriptor is lower than CPL). If the `int` instruction didn't require a privilege-level change, the x86 won't save `%ss` and `%esp`. After both cases, `%eip` is pointing to the address specified in the descriptor table, and the instruction at that address is the next instruction to be executed and the first instruction of the handler for `int` *n*. It is job of the operating system to implement these handlers, and below we will see what xv6 does.

An operating system can use the `iret` instruction to return from an `int` instruction. It pops the saved values during the `int` instruction from the stack, and resumes execution at the saved `%eip`.

Code: The first system call

The last chapter ended with `initcode.S` invoking a system call. Let's look at that

again (7513). The process pushed the arguments for an `exec` call on the process's stack, and put the system call number in `%eax`. The system call numbers match the entries in the syscalls array, a table of function pointers (3250). We need to arrange that the `int` instruction switches the processor from user mode to kernel mode, that the kernel invokes the right kernel function (i.e., `sys_exec`), and that the kernel can retrieve the arguments for `sys_exec`. The next few subsections describes how xv6 arranges this for system calls, and then we will discover that we can reuse the same code for interrupts and exceptions.

Code: Assembly trap handlers

Xv6 must set up the x86 hardware to do something sensible on encountering an `int` instruction, which causes the processor to generate a trap. The x86 allows for 256 different interrupts. Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses. Xv6 maps the 32 hardware interrupts to the range 32-63 and uses interrupt 64 as the system call interrupt.

`Tvinit` (2967), called from `main`, sets up the 256 entries in the table `idt`. Interrupt `i` is handled by the code at the address in `vectors[i]`. Each entry point is different, because the x86 provides does not provide the trap number to the interrupt handler. Using 256 different handlers is the only way to distinguish the 256 cases.

`Tvinit` handles `T_SYSCALL`, the user system call trap, specially: it specifies that the gate is of type "trap" by passing a value of `1` as second argument. Trap gates don't clear the `FL` flag, allowing other interrupts during the system call handler.

The kernel also sets the system call gate privilege to `DPL_USER`, which allows a user program to generate the trap with an explicit `int` instruction. xv6 doesn't allow processes to raise other interrupts (e.g., device interrupts) with `int`; if they try, they will encounter a general protection exception, which goes to vector 13.

When changing protection levels from user to kernel mode, the kernel shouldn't use the stack of the user process, because it may not be valid. The user process may be malicious or contain an error that causes the user `%esp` to contain an address that is not part of the process's user memory. Xv6 programs the x86 hardware to perform a stack switch on a trap by setting up a task segment descriptor through which the hardware loads a stack segment selector and a new value for `%esp`. The function `switchuvm` (1769) stores the address of the top of the kernel stack of the user process into the task segment descriptor.

When a trap occurs, the processor hardware does the following. If the processor was executing in user mode, it loads `%esp` and `%ss` from the task segment descriptor, pushes the old user `%ss` and `%esp` onto the new stack. If the processor was executing in kernel mode, none of the above happens. The processor then pushes the `%eflags`, `%cs`, and `%eip` registers. For some traps, the processor also pushes an error word. The processor then loads `%eip` and `%cs` from the relevant IDT entry.

xv6 uses a Perl script (2850) to generate the entry points that the IDT entries point to. Each entry pushes an error code if the processor didn't, pushes the interrupt number, and then jumps to `alltraps`.

`Alltraps` (2904) continues to save processor registers: it pushes `%ds`, `%es`, `%fs`,

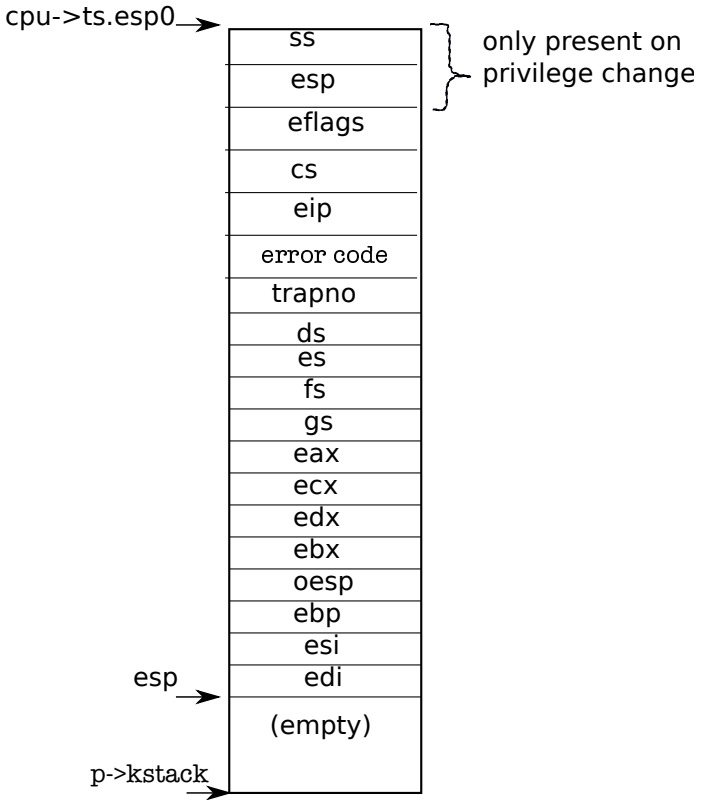

Figure 2-2. The trapframe on the kernel stack

`%gs`, and the general-purpose registers (2905-2910). The result of this effort is that the kernel stack now contains a `struct trapframe` (0602) containing the processor registers at the time of the trap (see Figure 2-2). The processor pushes `%ss`, `%esp`, `%eflags`, `%cs`, and `%eip`. The processor or the trap vector pushes an error number, and `alltraps` pushes the rest. The trap frame contains all the information necessary to restore the user mode processor registers when the kernel returns to the current process, so that the processor can continue exactly as it was when the trap started. Recall from Chapter 1, that `userinit` build a trapframe by hand to achieve this goal (see Figure 1-3).

In the case of the first system call, the saved `%eip` is the address of the instruction right after the `int` instruction. `%cs` is the user code segment selector. `%eflags` is the content of the eflags register at the point of executing the `int` instruction. As part of saving the general-purpose registers, `alltraps` also saves `%eax`, which contains the system call number for the kernel to inspect later.

Now that the user mode processor registers are saved, `alltraps` can finishing setting up the processor to run kernel C code. The processor set the selectors `%cs` and `%ss` before entering the handler; `alltraps` sets `%ds` and `%es` (2913-2915). It sets `%fs` and `%gs` to point at the `SEG_KCPU` per-CPU data segment (2916-2918).

Once the segments are set properly, `alltraps` can call the C trap handler `trap`. It pushes `%esp`, which points at the trap frame it just constructed, onto the stack as an argument to `trap` (2921). Then it calls `trap` (2922). After `trap` returns, `alltraps` pops

the argument off the stack by adding to the stack pointer (2923) and then starts executing the code at label `trapret`. We traced through this code in Chapter 1 when the first user process ran it to exit to user space. The same sequence happens here: popping through the trap frame restores the user mode registers and then `iret` jumps back into user space.

The discussion so far has talked about traps occurring in user mode, but traps can also happen while the kernel is executing. In that case the hardware does not switch stacks or save the stack pointer or stack segment selector; otherwise the same steps occur as in traps from user mode, and the same xv6 trap handling code executes. When `iret` later restores a kernel mode `%cs`, the processor continues executing in kernel mode.

Code: C trap handler

We saw in the last section that each handler sets up a trap frame and then calls the C function `trap`. `Trap` (3001) looks at the hardware trap number `tf->trapno` to decide why it has been called and what needs to be done. If the trap is `T_SYSCALL`, `trap` calls the system call handler `syscall`. We'll revisit the two `cp->killed` checks in Chapter 4.

After checking for a system call, trap looks for hardware interrupts (which we discuss below). In addition to the expected hardware devices, a trap can be caused by a spurious interrupt, an unwanted hardware interrupt.

If the trap is not a system call and not a hardware device looking for attention, `trap` assumes it was caused by incorrect behavior (e.g., divide by zero) as part of the code that was executing before the trap. If the code that caused the trap was a user program, xv6 prints details and then sets `cp->killed` to remember to clean up the user process. We will look at how xv6 does this cleanup in Chapter 4.

If it was the kernel running, there must be a kernel bug: `trap` prints details about the surprise and then calls `panic`.

Code: System calls

For system calls, `trap` invokes `syscall` (3275). `Syscall` loads the system call number from the trap frame, which contains the saved `%eax`, and indexes into the system call tables. For the first system call, `%eax` contains the value `SYS_exec` (3107), and `syscall` will invoke the `SYS_exec`'th entry of the system call table, which corresponds to invoking `sys_exec`.

`Syscall` records the return value of the system call function in `%eax`. When the trap returns to user space, it will load the values from `cp->tf` into the machine registers. Thus, when `exec` returns, it will return the value that the system call handler returned (3281). System calls conventionally return negative numbers to indicate errors, positive numbers for success. If the system call number is invalid, `syscall` prints an error and returns –1.

Later chapters will examine the implementation of particular system calls. This chapter is concerned with the mechanisms for system calls. There is one bit of mecha-

nism left: finding the system call arguments. The helper functions argint and argptr, argstr retrieve the *n*'th system call argument, as either an integer, pointer, or a string. `argint` uses the user-space `%esp` register to locate the *n'th* argument: `%esp` points at the return address for the system call stub. The arguments are right above it, at `%esp+4`. Then the nth argument is at `%esp+4+4*n`.

`argint` calls `fetchint` to read the value at that address from user memory and write it to `*ip`. `fetchint` can simply cast the address to a pointer, because the user and the kernel share the same page table, but the kernel must verify that the pointer by the user is indeed a pointer in the user part of the address space. The kernel has set up the page-table hardware to make sure that the process cannot access memory outside its local private memory: if a user program tries to read or write memory at an address of `p->sz` or above, the processor will cause a segmentation trap, and trap will kill the process, as we saw above. Now though, the kernel is running and it can derefence any address that the user might have passed, so it must check explicitly that the address is below `p->sz`

`argptr` is similar in purpose to `argint`: it interprets the *n*th system call argument. `argptr` calls `argint` to fetch the argument as an integer and then checks if the integer as a user pointer is indeed in the user part of the address space. Note that two checks occur during a call to code argptr . First, the user stack pointer is checked during the fetching of the argument. Then the argument, itself a user pointer, is checked.

`argstr` is the final member of the system call argument trio. It interprets the *n*th argument as a pointer. It ensures that the pointer points at a NUL-terminated string and that the complete string is located below the end of the user part of the address space.

The system call implementations (for example, sysproc.c and sysfile.c) are typically wrappers: they decode the arguments using `argint`, `argptr`, and `argstr` and then call the real implementations. In chapter 1, `sys_exec` uses these functions to get at its arguments.

Code: Interrupts

Devices on the motherboard can generate interrupts, and xv6 must setup the hardware to handle these interrupts. Without device support xv6 wouldn't be usable; a user couldn't type on the keyboard, a file system couldn't store data on disk, etc. Fortunately, adding interrupts and support for simple devices doesn't require much additional complexity. As we will see, interrupts can use the same code as for systems calls and exceptions.

Interrupts are similar to system calls, except devices generate them at any time. There is hardware on the motherboard to signal the CPU when a device needs attention (e.g., the user has typed a character on the keyboard). We must program the device to generate an interrupt, and arrange that a CPU receives the interrupt.

Let's look at the timer device and timer interrupts. We would like the timer hardware to generate an interrupt, say, 100 times per second so that the kernel can track the passage of time and so the kernel can time-slice among multiple running processes. The choice of 100 times per second allows for decent interactive performance

while not swamping the processor with handling interrupts.

Like the x86 processor itself, PC motherboards have evolved, and the way interrupts are provided has evolved too. The early boards had a simple programmable interrupt controler (called the PIC), and you can find the code to manage it in `picirq.c`.

With the advent of multiprocessor PC boards, a new way of handling interrupts was needed, because each CPU needs an interrupt controller to handle interrupts send to it, and there must be a method for routing interrupts to processors. This way consists of two parts: a part that is in the I/O system (the IO APIC, `ioapic.c`), and a part that is attached to each processor (the local APIC, `lapic.c`). Xv6 is designed for a board with multiple processors, and each processor must be programmed to receive interrupts.

To also work correctly on uniprocessors, Xv6 programs the programmable interrupt controler (PIC) (6732). Each PIC can handle a maximum of 8 interrupts (i.e., devices) and multiplex them on the interrupt pin of the processor. To allow for more than 8 devices, PICs can be cascaded and typically boards have at least two. Using `inb` and `outb` instructions Xv6 programs the master to generate IRQ 0 through 7 and the slave to generate IRQ 8 through 16. Initially xv6 programs the PIC to mask all interrupts. The code in `timer.c` sets timer 1 and enables the timer interrupt on the PIC (7374). This description omits some of the details of programming the PIC. These details of the PIC (and the IOAPIC and LAPIC) are not important to this text but the interested reader can consult the manuals for each device, which are referenced in the source files.

On multiprocessors, xv6 must program the IOAPIC, and the LAPIC on each processor. The IO APIC has a table and the processor can program entries in the table through memory-mapped I/O, instead of using `inb` and `outb` instructions. During initialization, xv6 programs to map interrupt 0 to IRQ 0, and so on, but disables them all. Specific devices enable particular interrupts and say to which processor the interrupt should be routed. For example, xv6 routes keyboard interrupts to processor 0 (7316). Xv6 routes disk interrupts to the highest numbered processor on the system, as we will see below.

The timer chip is inside the LAPIC, so that each processor can receive timer interrupts independently. Xv6 sets it up in `lapicinit` (6451). The key line is the one that programs the timer (6464). This line tells the LAPIC to periodically generate an interrupt at `IRQ_TIMER`, which is IRQ 0. Line (6493) enables interrupts on a CPU's LAPIC, which will cause it to deliver interrupts to the local processor.

A processor can control if it wants to receive interrupts through the `IF` flag in the eflags register. The instruction `cli` disables interrupts on the processor by clearing `IF`, and `sti` enables interrupts on a processor. Xv6 disables interrupts during booting of the main cpu (8212) and the other processors (1126). The scheduler on each processor enables interrupts (2414). To control that certain code fragments are not interrupted, xv6 disables interrupts during these code fragments (e.g., see `switchuvm` (1769)).

The timer interrupts through vector 32 (which xv6 chose to handle IRQ 0), which xv6 setup in `idtinit` (1265). The only difference between vector 32 and vector 64 (the one for system calls) is that vector 32 is an interrupt gate instead of a trap gate. Inter-

rupt gates clears IF, so that the interrupted processor doesn't receive interrupts while it is handling the current interrupt. From here on until trap, interrupts follow the same code path as system calls and exceptions, building up a trap frame.

Trap when it's called for a time interrupt, does just two things: increment the ticks variable (2963), and call wakeup. The latter, as we will see in Chapter 4, may cause the interrupt to return in a different process.

Drivers

A driver is the piece of code in an operating system that manage a particular device: it provides interrupt handlers for a device, causes a device to perform operations, causes a device to generate interrupts, etc. Driver code can be tricky to write because a driver executes concurrently with the device that it manages. In addition, the driver must understand the device's interface (e.g., which I/O ports do what), and that interface can be complex and poorly documented.

The disk driver provides a good example in xv6. The disk driver copies data from and back to the disk. Disk hardware traditionally presents the data on the disk as a numbered sequence of 512-byte *blocks* (also called *sectors*): sector 0 is the first 512 bytes, sector 1 is the next, and so on. To represent disk sectors an operating system has a structure that corresponds to one sector. The data stored in this structure is often out of sync with the disk: it might have not yet been read in from disk (the disk is working on it but hasn't returned the sector's content yet), or it might have been updated but not yet written out. The driver must ensure that the rest of xv6 doesn't get confused when the structure is out of sync with the disk.

Code: Disk driver

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

The disk driver represent disk sectors with a data structure called a buffer, struct buf (3400). Each buffer represents the contents of one sector on a particular disk device. The dev and sector fields give the device and sector number and the data field is an in-memory copy of the disk sector.

The flags track the relationship between memory and disk: the B_VALID flag means that data has been read in, and the B_DIRTY flag means that data needs to be written out. The B_BUSY flag is a lock bit; it indicates that some process is using the buffer and other processes must not. When a buffer has the B_BUSY flag set, we say the buffer is locked.

The kernel initializes the disk driver at boot time by calling ideinit (3701) from main (1232). Ideinit calls picenable and ioapicenable to enable the IDE_IRQ interrupt (3706-3707). The call to picenable enables the interrupt on a uniprocessor; ioapicenable enables the interrupt on a multiprocessor, but only on the last CPU (ncpu-1): on a two-processor system, CPU 1 handles disk interrupts.

Next, ideinit probes the disk hardware. It begins by calling idewait (3708) to wait for the disk to be able to accept commands. A PC motherboard presents the status bits of the disk hardware on I/O port 0x1f7. Idewait (3683) polls the status bits until the busy bit (IDE_BSY) is clear and the ready bit (IDE_DRDY) is set.

Now that the disk controller is ready, ideinit can check how many disks are present. It assumes that disk 0 is present, because the boot loader and the kernel were both loaded from disk 0, but it must check for disk 1. It writes to I/O port 0x1f6 to select disk 1 and then waits a while for the status bit to show that the disk is ready (3710-3717). If not, ideinit assumes the disk is absent.

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

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

Iderw (3804) adds the buffer b to the end of the queue (3817-3821). If the buffer is at the front of the queue, iderw must send it to the disk hardware by calling idestart (3774-3776); otherwise the buffer will be started once the buffers ahead of it are taken care of.

Idestart (3725) issues either a read or a write for the buffer's device and sector, according to the flags. If the operation is a write, idestart must supply the data now (3739) and the interrupt will signal that the data has been written to disk. If the operation is a read, the interrupt will signal that the data is ready, and the handler will read it. Note that iderw has detailed knowledge about the IDE device, and writes the right values at the right ports. If any of these outb statements is wrong, the IDE will do something differently than what we want. Getting these details right is one reason why writing device drivers is challenging.

Having added the request to the queue and started it if necessary, iderw must wait for the result. As discussed above, polling does not make efficient use of the CPU. Instead, iderw sleeps, waiting for the interrupt handler to record in the buffer's flags that the operation is done (3829-3830). While this process is sleeping, xv6 will schedule other processes to keep the CPU busy.

Eventually, the disk will finish its operation and trigger an interrupt. trap will call ideintr to handle it (3024). Ideintr (3752) consults the first buffer in the queue to find out which operation was happening. If the buffer was being read and the disk controller has data waiting, ideintr reads the data into the buffer with insl (3765-

3767). Now the buffer is ready: `ideintr` sets `B_VALID`, clears `B_DIRTY`, and wakes up any process sleeping on the buffer (3769-3772). Finally, `ideintr` must pass the next waiting buffer to the disk (3774-3776).

Real world

Supporting all the devices on a PC motherboard in its full glory is much work, because there are many devices, the devices have many features, and the protocol between device and driver can be complex. In many operating systems, the drivers together account for more code in the operating system than the core kernel.

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

Many operating systems have drivers for solid-state disks because they provide much faster access to data. But, although a solid-state works very differently from a traditional mechanical disk, both devices provide block-based interfaces and reading/writing blocks on a solid-state disk is still more expensive than reading/writing RAM.

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

Most of the devices in this chapter used I/O instructions to program them, which reflects the older nature of these devices. All modern devices are programmed using memory-mapped I/O.

Some drivers dynamically switch between polling and interrupts, because using interrupts can be expensive, but using polling can introduce delay until the driver processes an event. For example, for a network driver that receives a burst of packets, may switch from interrupts to polling since it knows that more packets must be processed and it is less expensive to process them using polling. Once no more packets need to be processed, the driver may switch back to interrupts, so that it will be alerted immediately when a new packet arrives.

The IDE driver routed interrupts statically to a particular processor. Some drivers have a sophisticated algorithm for routing interrupts to processor so that the load of processing packets is well balanced but good locality is achieved too. For example, a network driver might arrange to deliver interrupts for packets of one network connection to the processor that is managing that connection, while interrupts for packets of

another connection are delivered to another processor. This routing can get quite sophisticated; for example, if some network connections are short lived while others are long lived and the operating system wants to keep all processors busy to achieve high throughput.

If user process reads a file, the data for that file is copied twice. First, it is copied from the disk to kernel memory by the driver, and then later it is copied from kernel space to user space by the `read` system call. If the user process, then sends the data on the network, then the data is copied again twice: once from user space to kernel space and from kernel space to the network device. To support applications for which low latency is important (e.g., a Web serving static Web pages), operating systems use special code paths to avoid these many copies. As one example, in real-world operating systems, buffers typically match the hardware page size, so that read-only copies can be mapped into a process's address space using the paging hardware, without any copying.

Exercises

1. Set a breakpoint at the first instruction of syscall() to catch the very first system call (e.g., br syscall). What values are on the stack at this point? Explain the output of x/37x $esp at that breakpoint with each value labeled as to what it is (e.g., saved %ebp for trap, trapframe.eip, scratch space, etc.).

2. Add a new system call

3. Add a network driver

Chapter 3

Locking

Xv6 runs on multiprocessors, computers with multiple CPUs executing code independently. These multiple CPUs operate on a single physical address space and share data structures; xv6 must introduce a coordination mechanism to keep them from interfering with each other. Even on a uniprocessor, xv6 must use some mechanism to keep interrupt handlers from interfering with non-interrupt code. Xv6 uses the same low-level concept for both: a `lock`. A lock provides mutual exclusion, ensuring that only one CPU at a time can hold the lock. If xv6 only accesses a data structure while holding a particular lock, then xv6 can be sure that only one CPU at a time is accessing the data structure. In this situation, we say that the lock protects the data structure. The rest of this chapters why xv6 needs locks, how xv6 implements them, and how it uses them.

Race conditions

As an example on why we need locks, consider several processors sharing a single disk, such as the IDE disk in xv6. The disk driver maintains a linked list of the outstanding disk requests (3671) and processors may add new requests to the list concurrently (3804). If there were no concurrent requests, you might implement the linked list as follows:

```
1    struct list {
2      int data;
3      struct list *next;
4    };
5
6    struct list *list = 0;
7
8    void
9    insert(int data)
10   {
11     struct list *l;
12
13     l = malloc(sizeof *l);
14     l->data = data;
15     l->next = list;
16     list = l;
17   }
```

Proving this implementation correct is a typical exercise in a data structures and algorithms class. Even though this implementation can be proved correct, it isn't, at least not on a multiprocessor. If two different CPUs execute `insert` at the same time, it could happen that both execute line 15 before either executes 16 (see Figure 3-1). If

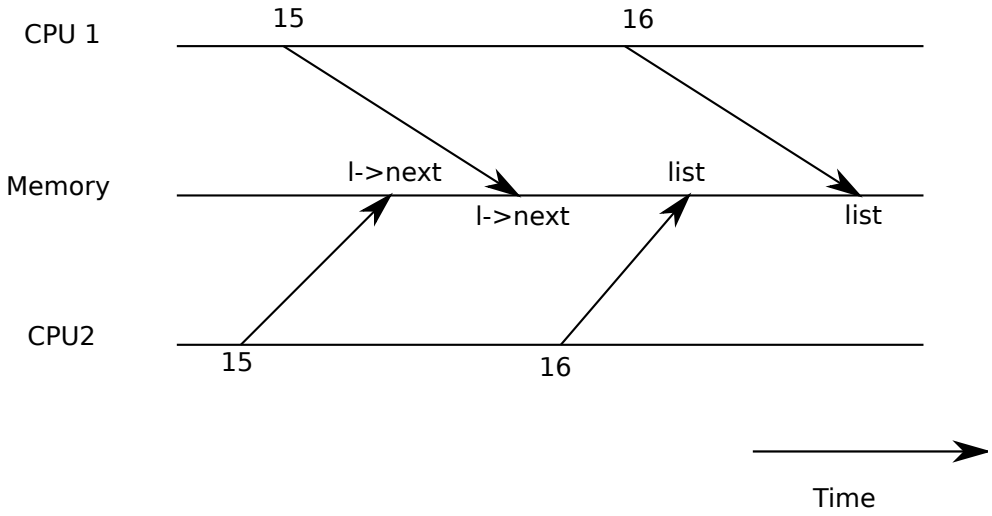

Figure 3-1. Example race

this happens, there will now be two list nodes with `next` set to the former value of `list`. When the two assignments to `list` happen at line 16, the second one will overwrite the first; the node involved in the first assignment will be lost. This kind of problem is called a `race condition`. The problem with races is that they depend on the exact timing of the two CPUs involved and how their memory operations are ordered by the memory system, and are consequently difficult to reproduce. For example, adding print statements while debugging `insert` might change the timing of the execution enough to make the race disappear.

The typical way to avoid races is to use a lock. Locks ensure mutual exclusion, so that only one CPU can execute `insert` at a time; this makes the scenario above impossible. The correctly locked version of the above code adds just a few lines (not numbered):

```
 6    struct list *list = 0;
      struct lock listlock;
 7
 8    void
 9    insert(int data)
10    {
11      struct list *l;
12
        acquire(&listlock);
13      l = malloc(sizeof *l);
14      l->data = data;
15      l->next = list;
16      list = l;
        release(&listlock);
17    }
```

When we say that a lock protects data, we really mean that the lock protects some collection of invariants that apply to the data. Invariants are properties of data structures that are maintained across operations. Typically, an operation's correct behavior

depends on the invariants being true when the operation begins. The operation may temporarily violate the invariants but must reestablish them before finishing. For example, in the linked list case, the invariant is that `list` points at the first node in the list and that each node's `next` field points at the next node. The implementation of `insert` vioilates this invariant temporarily: line 13 creates a new list element `l` with the intent that `l` be the first node in the list, but `l`'s next pointer does not point at the next node in the list yet (reestablished at line 15) and `list` does not point at `l` yet (reestablished at line 16). The race condition we examined above happened because a second CPU executed code that depended on the list invariants while they were (temporarily) violated. Proper use of a lock ensures that only one CPU at a time can operate on the data structure, so that no CPU will execute a data structure operation when the data structure's invariants do not hold.

Code: Locks

Xv6's represents a lock as a `struct spinlock` (1401). The critical field in the structure is `locked`, a word that is zero when the lock is available and non-zero when it is held. Logically, xv6 should acquire a lock by executing code like

```
21    void
22    acquire(struct spinlock *lk)
23    {
24      for(;;) {
25        if(!lk->locked) {
26          lk->locked = 1;
27          break;
28        }
29      }
30    }
```

Unfortunately, this implementation does not guarantee mutual exclusion on a modern multiprocessor. It could happen that two (or more) CPUs simultaneously reach line 25, see that `lk->locked` is zero, and then both grab the lock by executing lines 26 and 27. At this point, two different CPUs hold the lock, which violates the mutual exclusion property. Rather than helping us avoid race conditions, this implementation of `acquire` has its own race condition. The problem here is that lines 25 and 26 executed as separate actions. In order for the routine above to be correct, lines 25 and 26 must execute in one `atomic` (i.e., indivisible) step.

To execute those two lines atomically, xv6 relies on a special 386 hardware instruction, `xchg` (0569). In one atomic operation, `xchg` swaps a word in memory with the contents of a register. The function `acquire` (1474) repeats this `xchg` instruction in a loop; each iteration reads `lk->locked` and atomically sets it to 1 (1483). If the lock is held, `lk->locked` will already be 1, so the `xchg` returns 1 and the loop continues. If the `xchg` returns 0, however, `acquire` has successfully acquired the lock—`locked` was 0 and is now 1—so the loop can stop. Once the lock is acquired, `acquire` records, for debugging, the CPU and stack trace that acquired the lock. When a process acquires a lock and forget to release it, this information can help to identify the culprit. These debugging fields are protected by the lock and must only be edited while holding the

lock.

The function `release` (1502) is the opposite of `acquire`: it clears the debugging fields and then releases the lock.

Modularity and recursive locks

System design strives for clean, modular abstractions: it is best when a caller does not need to know how a callee implements particular functionality. Locks interfere with this modularity. For example, if a CPU holds a particular lock, it cannot call any function `f` that will try to reacquire that lock: since the caller can't release the lock until `f` returns, if `f` tries to acquire the same lock, it will spin forever, or deadlock.

There are no transparent solutions that allow the caller and callee to hide which locks they use. One common, transparent, but unsatisfactory solution is `recursive locks`, which allow a callee to reacquire a lock already held by its caller. The problem with this solution is that recursive locks can't be used to protect invariants. After `insert` called `acquire(&listlock)` above, it can assume that no other function holds the lock, that no other function is in the middle of a list operation, and most importantly that all the list invariants hold. In a system with recursive locks, `insert` can assume nothing after it calls `acquire`: perhaps `acquire` succeeded only because one of `insert`'s caller already held the lock and was in the middle of editing the list data structure. Maybe the invariants hold or maybe they don't. The list no longer protects them. Locks are just as important for protecting callers and callees from each other as they are for protecting different CPUs from each other; recursive locks give up that property.

Since there is no ideal transparent solution, we must consider locks part of the function's specification. The programmer must arrange that function doesn't invoke a function `f` while holding a lock that `f` needs. Locks force themselves into our abstractions.

Code: Using locks

Xv6 is carefully programmed with locks to avoid race conditions. A simple example is in the IDE driver (3650). As mentioned in the beginning of the chapter, `iderw` (3804) has a queue of disk requests and processors may add new requests to the list concurrently (3819). To protect this list and other invariants in the driver, `iderw` acquires the `idelock` (3815) and releases at the end of the function. Exercise 1 explores how to trigger the race condition that we saw at the beginning of the chapter by moving the `acquire` to after the queue manipulation. It is worthwhile to try the exercise because it will make clear that it is not that easy to trigger the race, suggesting that it is difficult to find race-conditions bugs. It is not unlikely that xv6 has some races.

A hard part about using locks is deciding how many locks to use and which data and invariants each lock protects. There are a few basic principles. First, any time a variable can be written by one CPU at the same time that another CPU can read or write it, a lock should be introduced to keep the two operations from overlapping. Second, remember that locks protect invariants: if an invariant involves multiple data

structures, typically all of the structures need to be protected by a single lock to ensure the invariant is maintained.

The rules above say when locks are necessary but say nothing about when locks are unnecessary, and it is important for efficiency not to lock too much, because locks reduce parallelism. If efficiency wasn't important, then one could use a uniprocessor computer and no worry at all about locks. For protecting kernel data structures, it would suffice to create a single lock that must be acquired on entering the kernel and released on exiting the kernel. Many uniprocessor operating systems have been converted to run on multiprocessors using this approach, sometimes called a "giant kernel lock," but the approach sacrifices true concurrency: only one CPU can execute in the kernel at a time. If the kernel does any heavy computation, it would be more efficient to use a larger set of more fine-grained locks, so that the kernel could execute on multiple CPUs simultaneously.

Ultimately, the choice of lock granularity is an exercise in parallel programming. Xv6 uses a few coarse data-structure specific locks; for example, xv6 uses a single lock protecting the process table and its invariants, which are described in Chapter 4. A more fine-grained approach would be to have a lock per entry in the process table so that threads working on different entries in the process table can proceed in parallel. However, it complicates operations that have invariants over the whole process table, since they might have to take out several locks. Hopefully, the examples of xv6 will help convey how to use locks.

Lock ordering

If a code path through the kernel must take out several locks, it is important that all code paths acquire the locks in the same order. If they don't, there is a risk of deadlock. Let's say two code paths in xv6 needs locks A and B, but code path 1 acquires locks in the order A and B, and the other code acquires them in the order B and A. This situation can result in a deadlock, because code path 1 might acquire lock A and before it acquires lock B, code path 2 might acquire lock B. Now neither code path can proceed, because code path 1 needs lock B, which code path 2 holds, and code path 2 needs lock A, which code path 1 holds. To avoid such deadlocks, all code paths must acquire locks in the same order. Deadlock avoidance is another example illustrating why locks must be part of a function's specification: the caller must invoke functions in a consistent order so that the functions acquire locks in the same order.

Because xv6 uses coarse-grained locks and xv6 is simple, xv6 has few lock-order chains. The longest chain is only two deep. For example, `ideintr` holds the ide lock while calling `wakeup`, which acquires the `ptable` lock. There are a number of other examples involving `sleep` and `wakeup`. These orderings come about because `sleep` and `wakeup` have a complicated invariant, as discussed in Chapter 4. In the file system there are a number of examples of chains of two because the file system must, for example, acquire a lock on a directory and the lock on a file in that directory to unlink a file from its parent directory correctly. Xv6 always acquires the locks in the order first parent directory and then the file.

Interrupt handlers

Xv6 uses locks to protect interrupt handlers running on one CPU from non-interrupt code accessing the same data on another CPU. For example, the timer interrupt handler (3014) increments `ticks` but another CPU might be in `sys_sleep` at the same time, using the variable (3373). The lock `tickslock` synchronizes access by the two CPUs to the single variable.

Interrupts can cause concurrency even on a single processor: if interrupts are enabled, kernel code can be stopped at any moment to run an interrupt handler instead. Suppose `iderw` held the `idelock` and then got interrupted to run `ideintr`. `Ideintr` would try to lock `idelock`, see it was held, and wait for it to be released. In this situation, `idelock` will never be released—only `iderw` can release it, and `iderw` will not continue running until `ideintr` returns—so the processor, and eventually the whole system, will deadlock.

To avoid this situation, if a lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled. Xv6 is more conservative: it never holds any lock with interrupts enabled. It uses `pushcli` (1555) and `popcli` (1566) to manage a stack of "disable interrupts" operations (`cli` is the x86 instruction that disables interrupts. `Acquire` calls `pushcli` before trying to acquire a lock (1476), and `release` calls `popcli` after releasing the lock (1521). `Pushcli` (1555) and `popcli` (1566) are more than just wrappers around `cli` and `sti`: they are counted, so that it takes two calls to `popcli` to undo two calls to `pushcli`; this way, if code acquires two different locks, interrupts will not be reenabled until both locks have been released.

It is important that `acquire` call `pushcli` before the `xchg` that might acquire the lock (1483). If the two were reversed, there would be a few instruction cycles when the lock was held with interrupts enabled, and an unfortunately timed interrupt would deadlock the system. Similarly, it is important that `release` call `popcli` only after the `xchg` that releases the lock (1483).

The interaction between interrupt handlers and non-interrupt code provides a nice example why recursive locks are problematic. If xv6 used recursive locks (a second acquire on a CPU is allowed if the first acquire happened on that CPU too), then interrupt handlers could run while non-interrupt code is in a critical section. This could create havoc, since when the interrupt handler runs, invariants that the handler relies on might be temporarily violated. For example, `ideintr` (3752) assumes that the linked list with outstanding requests is well-formed. If xv6 would have used recursive locks, then `ideintr` might run while `iderw` is in the middle of manipulating the linked list, and the linked list will end up in an incorrect state.

Memory ordering

This chapter has assumed that processors start and complete instructions in the order in which they appear in the program. Many processors, however, execute instructions out of order to achieve higher performance. If an instruction takes many cycles to complete, a processor may want to issue the instruction early so that it can overlap with other instructions and avoid processor stalls. For example, a processor may notice that in a serial sequence of instruction A and B are not dependent on each

other and start instruction B before A so that it will be completed when the processor completes A. Concurrency, however, may expose this reordering to software, which lead to incorrect behavior.

For example, one might wonder what happens if `release` just assigned 0 to `lk->locked`, instead of using `xchg`. The answer to this question is unclear, because different generations of x86 processors make different guarantees about memory ordering. If `lk->locked=0`, were allowed to be re-ordered say after `popcli`, than `acquire` might break, because to another thread interrupts would be enabled before a lock is released. To avoid relying on unclear processor specifications about memory ordering, xv6 takes no risk and uses `xchg`, which processors must guarantee not to reorder.

Real world

Concurrency primitives and parallel programming are active areas of of research, because programming with locks is still challenging. It is best to use locks as the base for higher-level constructs like synchronized queues, although xv6 does not do this. If you program with locks, it is wise to use a tool that attempts to identify race conditions, because it is easy to miss an invariant that requires a lock.

User-level programs need locks too, but in xv6 applications have one thread of execution and processes don't share memory, and so there is no need for locks in xv6 applications.

It is possible to implement locks without atomic instructions, but it is expensive, and most operating systems use atomic instructions.

Atomic instructions are not free either when a lock is contented. If one processor has a lock cached in its local cache, and another processor must acquire the lock, then the atomic instruction to update the line that holds the lock must move the line from the one processor's cache to the other processor's cache, and perhaps invalidate any other copies of the cache line. Fetching a cache line from another processor's cache can be orders of magnitude more expensive than fetching a line from a local cache.

To avoid the expenses associated with locks, many operating systems use lock-free data structures and algorithms, and try to avoid atomic operations in those algorithms. For example, it is possible to implemented a link list like the one in the beginning of the chapter that requires no locks during list searches, and one atomic instruction to insert an item in a list.

Exercises

1. get rid off the xchg in acquire. explain what happens when you run xv6?

2. move the acquire in iderw to before sleep. is there a race? why don't you observe it when booting xv6 and run stressfs? increase critical section with a dummy loop; what do you see now? explain.

3. do posted homework question.

4. Setting a bit in a buffer's `flags` is not an atomic operation: the processor makes a copy of `flags` in a register, edits the register, and writes it back. Thus it is important that two processes are not writing to `flags` at the same time. xv6 edits the `B_BUSY` bit only while holding `buflock` but edits the `B_VALID` and `B_WRITE` flags without holding any locks. Why is this safe?

Chapter 4

Scheduling

Any operating system is likely to run with more processes than the computer has processors, and so some plan is needed to time share the processors between the processes. An ideal plan is transparent to user processes. A common approach is to provide each process with the illusion that it has its own virtual processor, and have the operating system `multiplex` multiple virtual processors on a single physical processor. This chapter how xv6 multiplexes a processor among several processes.

Multiplexing

Xv6 adopts this multiplexing approach. When a process is waiting for disk request, xv6 puts it to sleep, and schedules another process to run. Furthermore, xv6 using timer interrupts to force a process to stop running on a processor after a fixed-amount of time (100 msec), so that it can schedule another process on the processor. This multiplexing creates the illusion that each process has its own CPU, just as xv6 used the memory allocator and hardware page tables to create the illusion that each process has its own memory.

Implementing multiplexing has a few challenges. First, how to switch from one process to another? Xv6 uses the standard mechanism of context switching; although the idea is simple, the code to implement is typically among the most opaque code in an operating system. Second, how to do context switching transparently? Xv6 uses the standard technique of using the timer interrupt handler to drive context switches. Third, many CPUs may be switching among processes concurrently, and a locking plan is necessary to avoid races. Fourth, when a process has exited its memory and other resources must be freed, but it cannot do all of this itself because (for example) it can't free its own kernel stack while still using it. Xv6 tries to solve these problems as simply as possible, but nevertheless the resulting code is tricky.

xv6 must provide ways for processes to coordinate among themselves. For example, a parent process may need to wait for one of its children to exit, or a process reading on a pipe may need to wait for some other process to write the pipe. Rather than make the waiting process waste CPU by repeatedly checking whether the desired event has happened, xv6 allows a process to give up the CPU and sleep waiting for an event, and allows another process to wake the first process up. Care is needed to avoid races that result in the loss of event notifications. As an example of these problems and their solution, this chapter examines the implementation of pipes.

Code: Context switching

As shown in Figure 4-1, to switch between processes, xv6 performs two kinds of

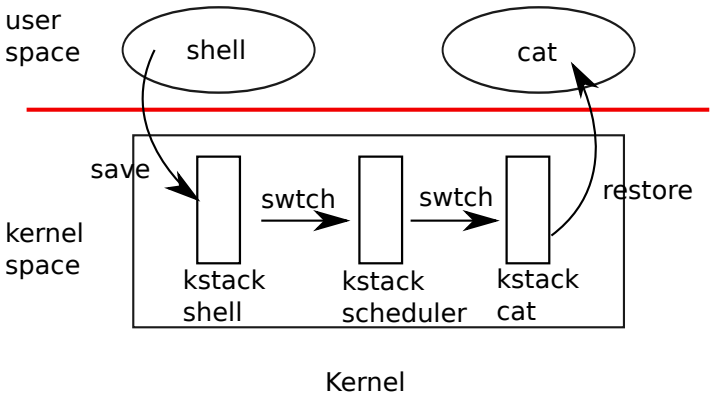

Figure 4-1. Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

---

context switches at a low level: from a process's kernel thread to the current CPU's scheduler thread, and from the scheduler thread to a process's kernel thread. xv6 never directly switches from one user-space process to another; this happens by way of a user-kernel transition (system call or interrupt), a context switch to the scheduler, a context switch to a new process's kernel thread, and a trap return. In this section we'll example the mechanics of switching between a kernel thread and a scheduler thread.

Every xv6 process has its own kernel stack and register set, as we saw in Chapter 1. Each CPU has a separate scheduler thread for use when it is executing the scheduler rather than any process's kernel thread. Switching from one thread to another involves saving the old thread's CPU registers, and restoring previously-saved registers of the new thread; the fact that `%esp` and `%eip` are saved and restored means that the CPU will switch stacks and switch what code it is executing.

`swtch` doesn't directly know about threads; it just saves and restores register sets, called `contexts`. When it is time for the process to give up the CPU, the process's kernel thread will call `swtch` to save its own context and return to the scheduler context. Each context is represented by a `struct context*`, a pointer to a structure stored on the kernel stack involved. `Swtch` takes two arguments: `struct context **old` and `struct context *new`. It pushes the current CPU register onto the stack and saves the stack pointer in `*old`. Then `swtch` copies `new` to `%esp`, pops previously saved registers, and returns.

Instead of following the scheduler into `swtch`, let's instead follow our user process back in. We saw in Chapter 2 that one possibility at the end of each interrupt is that `trap` calls `yield`. `Yield` in turn calls `sched`, which calls `swtch` to save the current context in `proc->context` and switch to the scheduler context previously saved in `cpu->scheduler` (2466).

`Swtch` (2652) starts by loading its arguments off the stack into the registers `%eax` and `%edx` (2659-2660); `swtch` must do this before it changes the stack pointer and can no longer access the arguments via `%esp`. Then `swtch` pushes the register state, creating a context structure on the current stack. Only the callee-save registers need to be saved; the convention on the x86 is that these are `%ebp`, `%ebx`, `%esi`, `%ebp`, and `%esp`.

Swtch pushes the first four explicitly (2663-2666); it saves the last implicitly as the struct context* written to *old (2669). There is one more important register: the program counter %eip was saved by the call instruction that invoked swtch and is on the stack just above %ebp. Having saved the old context, swtch is ready to restore the new one. It moves the pointer to the new context into the stack pointer (2670). The new stack has the same form as the old one that swtch just left—the new stack *was* the old one in a previous call to swtch—so swtch can invert the sequence to restore the new context. It pops the values for %edi, %esi, %ebx, and %ebp and then returns (2673-2677). Because swtch has changed the stack pointer, the values restored and the instruction address returned to are the ones from the new context.

In our example, sched called swtch to switch to cpu->scheduler, the per-CPU scheduler context. That context had been saved by scheduler's call to swtch (2428). When the swtch we have been tracing returns, it returns not to sched but to scheduler, and its stack pointer points at the current CPU's scheduler stack, not initproc's kernel stack.

Code: Scheduling

The last section looked at the low-level details of swtch; now let's take swtch as a given and examine the conventions involved in switching from process to scheduler and back to process. A process that wants to give up the CPU must acquire the process table lock ptable.lock, release any other locks it is holding, update its own state (proc->state), and then call sched. Yield (2472) follows this convention, as do sleep and exit, which we will examine later. Sched double-checks those conditions (2457-2462) and then an implication of those conditions: since a lock is held, the CPU should be running with interrupts disabled. Finally, sched calls swtch to save the current context in proc->context and switch to the scheduler context in cpu->scheduler. Swtch returns on the scheduler's stack as though scheduler's swtch had returned (2428). The scheduler continues the for loop, finds a process to run, switches to it, and the cycle repeats.

We just saw that xv6 holds ptable.lock across calls to swtch: the caller of swtch must already hold the lock, and control of the lock passes to the switched-to code. This convention is unusual with locks; the typical convention is the thread that acquires a lock is also responsible of releasing the lock, which makes it easier to reason about correctness. For context switching is necessary to break the typical convention because ptable.lock protects invariants on the process's state and context fields that are not true while executing in swtch. One example of a problem that could arise if ptable.lock were not held during swtch: a different CPU might decide to run the process after yield had set its state to RUNNABLE, but before swtch caused it to stop using its own kernel stack. The result would be two CPUs running on the same stack, which cannot be right.

A kernel thread always gives up its processor in sched and always switches to the same location in the scheduler, which (almost) always switches to a process in sched. Thus, if one were to print out the line numbers where xv6 switches threads, one would observe the following simple pattern: (2428), (2466), (2428), (2466), and so on. The proce-

dures in which this stylized switching between two threads happens are sometimes referred to as coroutines; in this example, sched and scheduler are co-routines of each other.

There is one case when the scheduler's swtch to a new process does not end up in sched. We saw this case in Chapter 1: when a new process is first scheduled, it begins at forkret (2483). Forkret exists only to honor this convention by releasing the ptable.lock; otherwise, the new process could start at trapret.

Scheduler (2408) runs a simple loop: find a process to run, run it until it stops, repeat. scheduler holds ptable.lock for most of its actions, but releases the lock (and explicitly enables interrupts) once in each iteration of its outer loop. This is important for the special case in which this CPU is idle (can find no RUNNABLE process). If an idling scheduler looped with the lock continuously held, no other CPU that was running a process could ever perform a context switch or any process-related system call, and in particular could never mark a process as RUNNABLE so as to break the idling CPU out of its scheduling loop. The reason to enable interrupts periodically on an idling CPU is that there might be no RUNNABLE process because processes (e.g., the shell) are waiting for I/O; if the scheduler left interrupts disabled all the time, the I/O would never arrive.

The scheduler loops over the process table looking for a runnable process, one that has p->state == RUNNABLE. Once it finds a process, it sets the per-CPU current process variable proc, switches to the process's page table with switchuvm, marks the process as RUNNING, and then calls swtch to start running it (2422-2428).

One way to think about the structure of the scheduling code is that it arranges to enforce a set of invariants about each process, and holds ptable.lock whenever those invariants are not true. One invariant is that if a process is RUNNING, things must be set up so that a timer interrupt's yield can correctly switch away from the process; this means that the CPU registers must hold the process's register values (i.e. they aren't actually in a context), %cr3 must refer to the process's pagetable, %esp must refer to the process's kernel stack so that swtch can push registers correctly, and proc must refer to the process's proc[] slot. Another invariant is that if a process is RUNNABLE, things must be set up so that an idle CPU's scheduler can run it; this means that p->context must hold the process's kernel thread variables, that no CPU is executing on the process's kernel stack, that no CPU's %cr3 refers to the process's page table, and that no CPU's proc refers to the process.

Maintaining the above invariants is the reason why xv6 acquires ptable.lock in one thread (often in yield) and releases the lock in a different thread (the scheduler thread or another next kernel thread). Once the code has started to modify a running process's state to make it RUNNABLE, it must hold the lock until it has finished restoring the invariants: the earliest correct release point is after scheduler stops using the process's page table and clears proc. Similarly, once scheduler starts to convert a runnable process to RUNNING, the lock cannot be released until the kernel thread is completely running (after the swtch, e.g. in yield).

ptable.lock protects other things as well: allocation of process IDs and free process table slots, the interplay between exit and wait, the machinery to avoid lost wakeups (see next section), and probably other things too. It might be worth thinking

about whether the different functions of `ptable.lock` could be split up, certainly for clarity and perhaps for performance.

Sleep and wakeup

Locks help CPUs and processes avoid interfering with each other, and scheduling helps processes share a CPU, but so far we have no abstractions that make it easy for processes to communicate. Sleep and wakeup fill that void, allowing one process to sleep waiting for an event and another process to wake it up once the event has happened. Sleep and wakeup are often called `sequence coordination` or `conditional synchronization` mechanisms, and there are many other such mechanisms in the operating systems literature.

To illustrate what we mean, let's consider a simple producer/consumer queue. This queue is similar to the one used by the IDE driver to synchronize a processor and device driver (see Chapter 2), but abstracts all IDE-specific code away. The queue allows one process to send a nonzero pointer to another process. Assuming there is only one sender and one receiver and they execute on different CPUs, this implementation is correct:

```
100    struct q {
101      void *ptr;
102    };
103
104    void*
105    send(struct q *q, void *p)
106    {
107      while(q->ptr != 0)
108        ;
109      q->ptr = p;
110    }
111
112    void*
113    recv(struct q *q)
114    {
115      void *p;
116
117      while((p = q->ptr) == 0)
118        ;
119      q->ptr = 0;
120      return p;
121    }
```

`Send` loops until the queue is empty (`ptr == 0`) and then puts the pointer `p` in the queue. `Recv` loops until the queue is non-empty and takes the pointer out. When run in different processes, `send` and `recv` both edit `q->ptr`, but `send` only writes to the pointer when it is zero and `recv` only writes to the pointer when it is nonzero, so they do not step on each other.

The implementation above may be correct, but it is expensive. If the sender sends rarely, the receiver will spend most of its time spinning in the `while` loop hoping for a pointer. The receiver's CPU could find more productive work if there were a

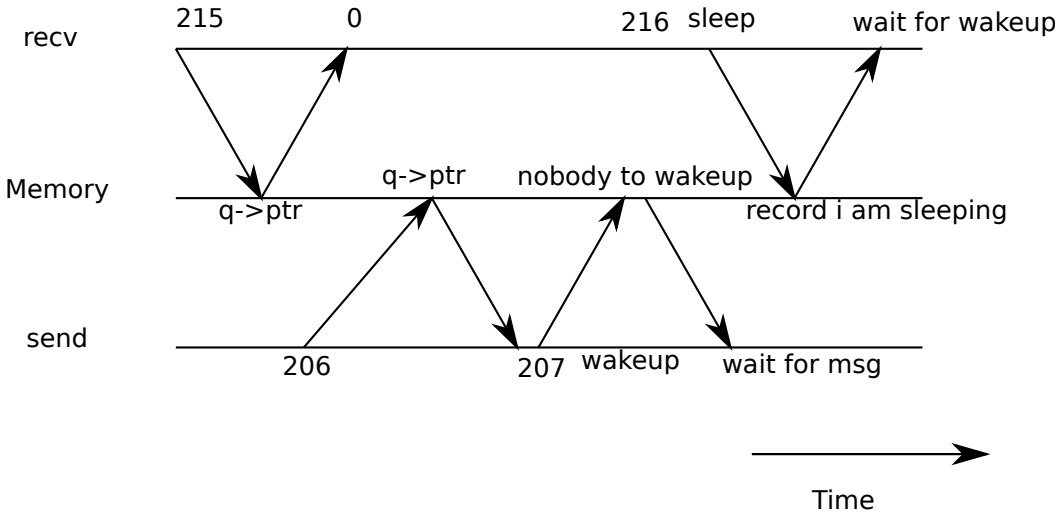

Figure 4-2. Example lost wakeup problem

way for the receiver to be notified when the `send` had delivered a pointer.

Let's imagine a pair of calls, `sleep` and `wakeup`, that work as follows. `Sleep(chan)` sleeps on the arbitrary value `chan`, called the `wait channel`. `Sleep` puts the calling process to sleep, releasing the CPU for other work. `Wakeup(chan)` wakes all processes sleeping on `chan` (if any), causing their `sleep` calls to return. If no processes are waiting on `chan`, `wakeup` does nothing. We can change the queue implementation to use `sleep` and `wakeup`:

```
201    void*
202    send(struct q *q, void *p)
203    {
204      while(q->ptr != 0)
205        ;
206      q->ptr = p;
207      wakeup(q);  /* wake recv */
208    }
209
210    void*
211    recv(struct q *q)
212    {
213      void *p;
214
215      while((p = q->ptr) == 0)
216        sleep(q);
217      q->ptr = 0;
218      return p;
219    }
```

`Recv` now gives up the CPU instead of spinning, which is nice. However, it turns out not to be straightforward to design `sleep` and `wakeup` with this interface without suffering from what is known as the "lost wake up" problem (see Figure 4-2). Suppose that `recv` finds that `q->ptr == 0` on line 215 and decides to call `sleep`. Before `recv`

can sleep, `send` runs on another CPU: it changes `q->ptr` to be nonzero and calls `wakeup`, which finds no processes sleeping and thus does nothing. Now `recv` continues executing at line 216: it calls `sleep` and goes to sleep. This causes a problem: `recv` is asleep waiting for a pointer that has already arrived. The next `send` will sleep waiting for `recv` to consume the pointer in the queue, at which point the system will be `deadlocked`.

The root of this problem is that the invariant that `recv` only sleeps when `q->ptr == 0` is violated by `send` running at just the wrong moment. To protect this invariant, we introduce a lock, which `sleep` releases only after the calling process is asleep; this avoids the missed wakeup in the example above. Once the calling process is awake again `sleep` reacquires the lock before returning. We would like to be able to have the following code:

```
300    struct q {
301      struct spinlock lock;
302      void *ptr;
303    };
304
305    void*
306    send(struct q *q, void *p)
307    {
308      acquire(&q->lock);
309      while(q->ptr != 0)
310        ;
311      q->ptr = p;
312      wakeup(q);
313      release(&q->lock);
314    }
315
316    void*
317    recv(struct q *q)
318    {
319      void *p;
320
321      acquire(&q->lock);
322      while((p = q->ptr) == 0)
323        sleep(q, &q->lock);
324      q->ptr = 0;
325      release(&q->lock);
326      return p;
327    }
```

The fact that `recv` holds `q->lock` prevents `send` from trying to wake it up between `recv`'s check of `q->ptr` and its call to `sleep`. Of course, the receiving process had better not hold `q->lock` while it is sleeping, since that would prevent the sender from waking it up, and lead to deadlock. So what we want is for sleep to atomically release `q->lock` and put the receiving process to sleep.

A complete sender/receiver implementation would also sleep in `send` when waiting for a receiver to consume the value from a previous `send`.

Code: Sleep and wakeup

Let's look at the implementation of `sleep` and `wakeup` in xv6. The basic idea is to have `sleep` mark the current process as `SLEEPING` and then call `sched` to release the processor; `wakeup` looks for a process sleeping on the given pointer and marks it as `RUNNABLE`.

`Sleep` (2503) begins with a few sanity checks: there must be a current process (2505) and `sleep` must have been passed a lock (2508-2509). Then `sleep` acquires `ptable.lock` (2518). Now the process going to sleep holds both `ptable.lock` and `lk`. Holding `lk` was necessary in the caller (in the example, `recv`): it ensured that no other process (in the example, one running `send`) could start a call `wakeup(chan)`. Now that `sleep` holds `ptable.lock`, it is safe to release `lk`: some other process may start a call to `wakeup(chan)`, but `wakeup` will not run until it can acquire `ptable.lock`, so it must wait until `sleep` has finished putting the process to sleep, keeping the `wakeup` from missing the `sleep`.

There is a minor complication: if `lk` is equal to `&ptable.lock`, then `sleep` would deadlock trying to acquire it as `&ptable.lock` and then release it as `lk`. In this case, `sleep` considers the acquire and release to cancel each other out and skips them entirely (2517).

Now that `sleep` holds `ptable.lock` and no others, it can put the process to sleep by recording the sleep channel, changing the process state, and calling `sched` (2523-2525).

At some point later, a process will call `wakeup(chan)`. `Wakeup` (2553) acquires `ptable.lock` and calls `wakeup1`, which does the real work. It is important that `wakeup` hold the `ptable.lock` both because it is manipulating process states and because, as we just saw, `ptable.lock` makes sure that `sleep` and `wakeup` do not miss each other. `Wakeup1` is a separate function because sometimes the scheduler needs to execute a wakeup when it already holds the `ptable.lock`; we will see an example of this later. `Wakeup1` (2553) loops over the process table. When it finds a process in state `SLEEPING` with a matching `chan`, it changes that process's state to `RUNNABLE`. The next time the scheduler runs, it will see that the process is ready to be run.

`Wakeup` must always be called while holding a lock that prevents observation of whatever the wakeup condition is; in the example above that lock is `q->lock`. The complete argument for why the sleeping process won't miss a wakeup is that at all times from before it checks the condition until after it is asleep, it holds either the lock on the condition or the `ptable.lock` or both. Since `wakeup` executes while holding both of those locks, the wakeup must execute either before the potential sleeper checks the condition, or after the potential sleeper has completed putting itself to sleep.

It is sometimes the case that multiple processes are sleeping on the same channel; for example, more than one process trying to read from a pipe. A single call to `wakeup` will wake them all up. One of them will run first and acquire the lock that `sleep` was called with, and (in the case of pipes) read whatever data is waiting in the pipe. The other processes will find that, despite being woken up, there is no data to be read. From their point of view the wakeup was "spurious," and they must sleep again. For this reason sleep is always called inside a loop that checks the condition.

Callers of sleep and wakeup can use any mutually convenient number as the channel; in practice xv6 often uses the address of a kernel data structure involved in

the waiting, such as a disk buffer. No harm is done if two uses of sleep/wakeup accidentally choose the same channel: they will see spurious wakeups, but looping as described above will tolerate this problem. Much of the charm of sleep/wakeup is that it is both lightweight (no need to create special data structures to act as sleep channels) and provides a layer of indirection (callers need not know what specific process they are interacting with).

Code: Pipes

The simple queue we used earlier in this chapter was a toy, but xv6 contains two real queues that uses `sleep` and `wakeup` to synchronize readers and writers. One is in the IDE driver: processes add a disk requests to a queue and then calls `sleep`. The interrupt handler uses `wakeup` to alert the process that its request has completed.

An more complex example is the implementation of pipes. We saw the interface for pipes in Chapter 0: bytes written to one end of a pipe are copied in an in-kernel buffer and then can be read out of the other end of the pipe. Future chapters will examine the file system support surrounding pipes, but let's look now at the implementations of `pipewrite` and `piperead`.

Each pipe is represented by a `struct pipe`, which contains a `lock` and a `data` buffer. The fields `nread` and `nwrite` count the number of bytes read from and written to the buffer. The buffer wraps around: the next byte written after `buf[PIPESIZE-1]` is `buf[0]`, but the counts do not wrap. This convention lets the implementation distinguish a full buffer (`nwrite == nread+PIPESIZE`) from an empty buffer `nwrite == nread`), but it means that indexing into the buffer must use `buf[nread % PIPESIZE]` instead of just `buf[nread]` (and similarly for `nwrite`). Let's suppose that calls to `piperead` and `pipewrite` happen simultaneously on two different CPUs.

`Pipewrite` (5880) begins by acquiring the pipe's lock, which protects the counts, the data, and their associated invariants. `Piperead` (5901) then tries to acquire the lock too, but cannot. It spins in `acquire` (1474) waiting for the lock. While `piperead` waits, `pipewrite` loops over the bytes being written—`addr[0]`, `addr[1]`, ..., `addr[n-1]`—adding each to the pipe in turn (5894). During this loop, it could happen that the buffer fills (5886). In this case, `pipewrite` calls `wakeup` to alert any sleeping readers to the fact that there is data waiting in the buffer and then sleeps on `&p->nwrite` to wait for a reader to take some bytes out of the buffer. `Sleep` releases `p->lock` as part of putting `pipewrite`'s process to sleep.

Now that `p->lock` is available, `piperead` manages to acquire it and start running in earnest: it finds that `p->nread != p->nwrite` (5906) (`pipewrite` went to sleep because `p->nwrite == p->nread+PIPESIZE` (5886)) so it falls through to the `for` loop, copies data out of the pipe (5913-5917), and increments `nread` by the number of bytes copied. That many bytes are now available for writing, so `piperead` calls `wakeup` (5918) to wake any sleeping writers before it returns to its caller. `Wakeup` finds a process sleeping on `&p->nwrite`, the process that was running `pipewrite` but stopped when the buffer filled. It marks that process as `RUNNABLE`.

The pipe code uses separate sleep channels for reader and writer (`p->nread` and `p->nwrite`); this might make the system more efficient in the unlikely event that there

are lots of readers and writers waiting for the same pipe. The pipe code sleeps inside a loop checking the sleep condition; if there are multiple readers or writers, all but the first process to wake up will see the condition is still false and sleep again.

Code: Wait and exit

`Sleep` and `wakeup` can be used in many kinds of situations involving a condition that can be checked needs to be waited for. As we saw in Chapter 0, a parent process can call `wait` to wait for a child to exit. In xv6, when a child exits, it does not die immediately. Instead, it switches to the `ZOMBIE` process state until the parent calls `wait` to learn of the exit. The parent is then responsible for freeing the memory associated with the process and preparing the `struct proc` for reuse. Each process structure keeps a pointer to its parent in `p->parent`. If the parent exits before the child, the initial process `init` adopts the child and waits for it. This step is necessary to make sure that some process cleans up after the child when it exits. All the process structures are protected by `ptable.lock`.

`Wait` begins by acquiring `ptable.lock`. Then it scans the process table looking for children. If `wait` finds that the current process has children but that none of them have exited, it calls `sleep` to wait for one of the children to exit (2389) and loops. Here, the lock being released in `sleep` is `ptable.lock`, the special case we saw above.

`Exit` acquires `ptable.lock` and then wakes the current process's parent (2326). This may look premature, since `exit` has not marked the current process as a `ZOMBIE` yet, but it is safe: although the parent is now marked as `RUNNABLE`, the loop in `wait` cannot run until `exit` releases `ptable.lock` by calling `sched` to enter the scheduler, so `wait` can't look at the exiting process until after the state has been set to `ZOMBIE` (2338). Before exit reschedules, it reparents all of the exiting process's children, passing them to the `initproc` (2328-2335). Finally, `exit` calls `sched` to relinquish the CPU.

Now the scheduler can choose to run the exiting process's parent, which is asleep in `wait` (2389). The call to `sleep` returns holding `ptable.lock`; `wait` rescans the process table and finds the exited child with `state == ZOMBIE`. (2332). It records the child's `pid` and then cleans up the `struct proc`, freeing the memory associated with the process (2368-2376).

The child process could have done most of the cleanup during `exit`, but it is important that the parent process be the one to free `p->kstack` and `p->pgdir`: when the child runs `exit`, its stack sits in the memory allocated as `p->kstack` and it uses its own pagetable. They can only be freed after the child process has finished running for the last time by calling `swtch` (via `sched`). This is one reason that the scheduler procedure runs on its own stack rather than on the stack of the thread that called `sched`.

Real world

The xv6 scheduler implements a simple scheduling policy, which runs each process in turn. This policy is called `round robin`. Real operating systems implement more sophisticated policies that, for example, allow processes to have priorities. The idea is that a runnable high-priority process will be preferred by the scheduler over a

runnable low-priority thread. These policies can become complex quickly because there are often competing goals: for example, the operating might also want to guarantee fairness and high-throughput. In addition, complex policies may lead to unintended interactions such as `priority inversion` and `convoys`. Priority inversion can happen when a low-priority and high-priority process share a lock, which when acquired by the low-priority process can cause the high-priority process to not run. A long convoy can form when many high-priority processes are waiting for a low-priority process that acquires a shared lock; once a convoy has formed they can persist for long period of time. To avoid these kinds of problems additional mechanisms are necessary in sophisticated schedulers.

`Sleep` and `wakeup` are a simple and effective synchronization method, but there are many others. The first challenge in all of them is to avoid the "missed wakeups" problem we saw at the beginning of the chapter. The original Unix kernel's `sleep` simply disabled interrupts, which sufficed because Unix ran on a single-CPU system. Because xv6 runs on multiprocessors, it adds an explicit lock to `sleep`. FreeBSD's `msleep` takes the same approach. Plan 9's `sleep` uses a callback function that runs with the scheduling lock held just before going to sleep; the function serves as a last minute check of the sleep condition, to avoid missed wakeups. The Linux kernel's `sleep` uses an explicit process queue instead of a wait channel; the queue has its own internal lock.

Scanning the entire process list in `wakeup` for processes with a matching `chan` is inefficient. A better solution is to replace the `chan` in both `sleep` and `wakeup` with a data structure that holds a list of processes sleeping on that structure. Plan 9's `sleep` and `wakeup` call that structure a rendezvous point or `Rendez`. Many thread libraries refer to the same structure as a condition variable; in that context, the operations `sleep` and `wakeup` are called `wait` and `signal`. All of these mechanisms share the same flavor: the sleep condition is protected by some kind of lock dropped atomically during sleep.

The implementation of `wakeup` wakes up all processes that are waiting on a particular channel, and it might be the case that many processes are waiting for that particular channel. The operating system will schedules all these processes and they will race to check the sleep condition. Processes that behave in this way are sometimes called a `thundering herd`, and it is best avoided. Most condition variables have two primitives for `wakeup`: `signal`, which wakes up one process, and `broadcast`, which wakes up all processes waiting.

Semaphores are another common coordination mechanism. A semaphore is an integer value with two operations, increment and decrement (or up and down). It is aways possible to increment a semaphore, but the semaphore value is not allowed to drop below zero: a decrement of a zero semaphore sleeps until another process increments the semaphore, and then those two operations cancel out. The integer value typically corresponds to a real count, such as the number of bytes available in a pipe buffer or the number of zombie children that a process has. Using an explicit count as part of the abstraction avoids the "missed wakeup" problem: there is an explicit count of the number of wakeups that have occurred. The count also avoids the spurious wakeup and thundering herd problems.

Exercises

1. Sleep has to check lk != &ptable.lock to avoid a deadlock (2517-2520). It could eliminate the special case by replacing

```
if(lk != &ptable.lock){
  acquire(&ptable.lock);
  release(lk);
}
```

with

```
release(lk);
acquire(&ptable.lock);
```

Doing this would break `sleep`. How?

2. Most process cleanup could be done by either `exit` or `wait`, but we saw above that `exit` must not free `p->stack`. It turns out that `exit` must be the one to close the open files. Why? The answer involves pipes.

3. Implement semaphores in xv6. You can use mutexes but do not use sleep and wakeup. Replace the uses of sleep and wakeup in xv6 with semaphores. Judge the result.

Chapter 5

File system

The purpose of a file system is to organize and store data. File systems typically support sharing of data among users and applications, as well `persistence` so that data is still available after a reboot.

The xv6 file system provides Unix-like files, directories, and pathnames (see Chapter 0), and stores its data on an IDE disk for persistence (see Chapter 2). The file system addresses several challenges:

- The file system needs on-disk data structures to represent the tree of named directories and files, to record the identities of the blocks that hold each file's content, and to record which areas of the disk are free.
- The file system must support `crash recovery`. That is, if a crash (e.g., power failure) occurs, the file system must still work correctly after a restart. The risk is that a crash might interrupt a sequence of updates and leave inconsistent on-disk data structures (e.g., a block that is both used in a file and marked free).
- Different processes may operate on the file system at the same time, and must coordinate to maintain invariants.
- Accessing a disk is orders of magnitude slower than accessing memory, so the file system must maintain an in-memory cache of popular blocks.

The rest of this chapter explains how xv6 addresses these challenges.

Overview

The xv6 file system implementation is organized in 6 layers, as shown in Figure 5-1. The lowest layer reads and writes blocks on the IDE disk through the buffer cache, which synchronizes access to disk blocks, making sure that only one kernel process at a time can edit the file system data stored in any particular block. The second layer allows higher layers to wrap updates to several blocks in a `transaction`, to ensure that the blocks are updated atomically (i.e., all of them are updated or none). The third layer provides unnamed files, each represented using an `inode` and a sequence of blocks holding the file's data. The fourth layer implements directories as a special kind of inode whose content is a sequence of directory entries, each of which contains a name and a reference to the named file's inode. The fifth layer provides hierarchical path names like `/usr/rtm/xv6/fs.c`, using recursive lookup. The final layer abstracts many Unix resources (e.g., pipes, devices, files, etc.) using the file system interface, simplifying the lives of application programmers.

The file system must have a plan for where it stores inodes and content blocks on the disk. To do so, xv6 divides the disk into several sections, as shown in Figure 5-2. The file system does not use block 0 (it holds the boot sector). Block 1 is called the `superblock`; it contains metadata about the file system (the file system size in blocks,

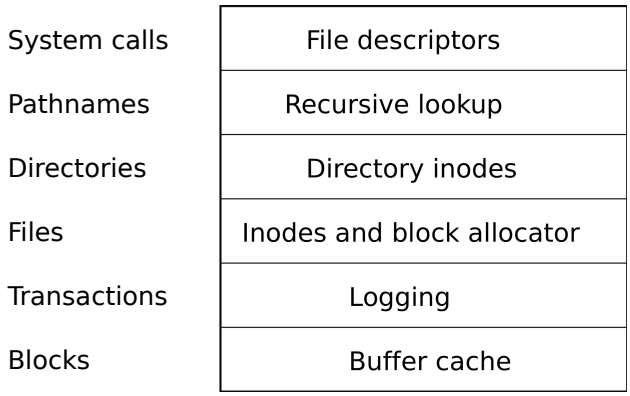

Figure 5-1. Layers of the xv6 file system.

the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold inodes, with multiple inodes per block. After those come bitmap blocks tracking which data blocks are in use (i.e., which are used in of some file). Most of the remaining blocks are data blocks, which hold file and directory contents. The blocks at the very end of the disk hold the log for transactions.

The rest of this chapter discusses each layer, starting from the bottom. Look out for situations where well-chosen abstractions at lower layers ease the design of higher ones. The file system is a good example of how well-designed abstractions lead to surprising generality.

Buffer cache Layer

The buffer cache has two jobs: (1) synchronize access to disk blocks to ensure that only one copy of a block is in memory and that only one kernel thread at a time uses that copy; (2) cache popular blocks so that they don't to be re-read from the slow disk. The code is in `bio.c`.

The main interface exported by the buffer cache consists of `bread` and `bwrite`; the former obtains a `buffer` containing a copy of a block which can be read or modified in memory, and the latter writes a modified buffer to the appropriate block on the disk. A kernel thread must release a buffer by calling `brelse` when it is done with it.

The buffer cache synchronizes access to each block by allowing at most one kernel thread to have a reference to the block's buffer. If one kernel thread has obtained a reference to a buffer but hasn't yet released it, other threads' calls to `bread` for the same block will wait. Higher file system layers rely on the buffer cache's block sychronization to help them maintain invariants.

The buffer cache has a fixed number of buffers to hold disk blocks, which means that if the file system asks for a block that is not already in the cache, the buffer cache must recycle a buffer currently holding some other block. The buffer cache recycles the least recently used buffer for the new block. The assumption is that the least recently used buffer is the one least likely to be used again soon.

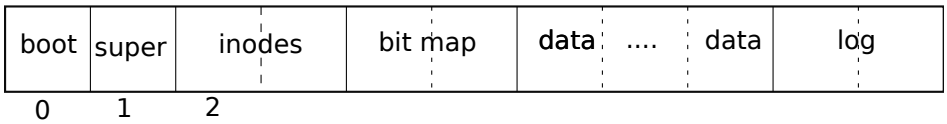

Figure 5-2. Structure of the xv6 file system. The header `fs.h` (3550) contains constants and data structures describing the exact layout of the file system.

Code: Buffer cache

The buffer cache is a doubly-linked list of buffers. The function `binit`, called by `main` (1229), initializes the list with the `NBUF` buffers in the static array `buf` (3900-3909). All other access to the buffer cache refers to the linked list via `bcache.head`, not the `buf` array.

A buffer has three state bits associated with it. `B_VALID` indicates that the buffer contains a valid copy of the block. `B_DIRTY` indicates that the buffer content has been modified and needs to be written to the disk. `B_BUSY` indicates that some kernel thread has a reference to this buffer and has not yet released it.

`Bread` (3952) calls `bget` to get a buffer for the given sector (3956). If the buffer needs to be read from disk, `bread` calls `iderw` to do that before returning the buffer.

`Bget` (3916) scans the buffer list for a buffer with the given device and sector numbers (3923-3934). If there is such a buffer, and the buffer is not busy, `bget` sets the `B_BUSY` flag and returns (3926-3933). If the buffer is already in use, `bget` sleeps on the buffer to wait for its release. When `sleep` returns, `bget` cannot assume that the buffer is now available. In fact, since `sleep` released and reacquired `buf_table_lock`, there is no guarantee that `b` is still the right buffer: maybe it has been reused for a different disk sector. `Bget` has no choice but to start over (3932), hoping that the outcome will be different this time.

If `bget` didn't have the `goto` statement, then the race in Figure 5-3 could occur. The first process has a buffer and has loaded sector 3 in it. Now two other processes come along. The first one does a `get` for buffer 3 and sleeps in the loop for cached blocks. The second one does a `get` for buffer 4, and could sleep on the same buffer but in the loop for freshly allocated blocks because there are no free buffers and the buffer that holds 3 is the one at the front of the list and is selected for reuse. The first process releases the buffer and `wakeup` happens to schedule process 3 first, and it will grab the buffer and load sector 4 in it. When it is done it will release the buffer (containing sector 4) and wakeup process 2. Without the `goto` statement process 2 will mark the buffer `BUSY`, and return from `bget`, but the buffer contains sector 4, instead of 3. This error could result in all kinds of havoc, because sectors 3 and 4 have different content; xv6 uses them for storing inodes.

If there is no buffer for the given sector, `bget` must make one, possibly reusing a buffer that held a different sector. It scans the buffer list a second time, looking for a block that is not busy: any such block can be used (3936-3938). `Bget` edits the block metadata to record the new device and sector number and mark the block busy before returning the block (3941-3943). Note that the assignment to `flags` not only sets the `B_BUSY` bit but also clears the `B_VALID` and `B_DIRTY` bits, making sure that `bread` will

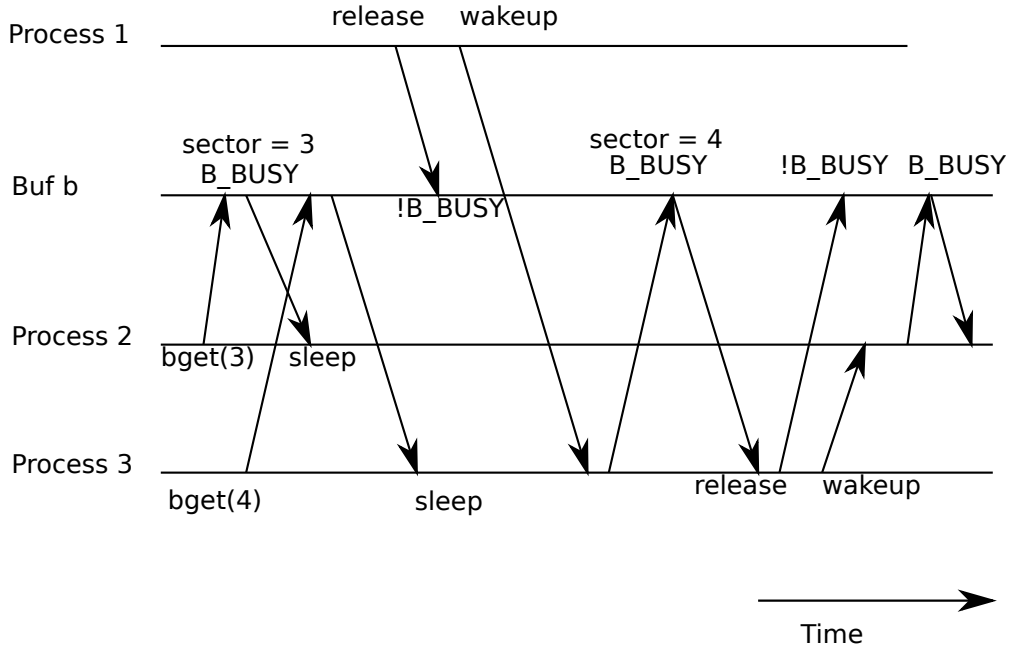

Figure 5-3. A race resulting in process 3 receiving a buffer containing block 4, even though it asked for block 3.

---

refresh the buffer data from disk rather than use the previous block's contents.

Because the buffer cache is used for synchronization, it is important that there is only ever one buffer for a particular disk sector. The assignments (3939-3941) are only safe because `bget`'s first loop determined that no buffer already existed for that sector, and `bget` has not given up `buf_table_lock` since then.

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

Once `bread` has returned a buffer to its caller, the caller has exclusive use of the buffer and can read or write the data bytes. If the caller does write to the data, it must call `bwrite` to write the changed data to disk before releasing the buffer. `Bwrite` (3964) sets the `B_DIRTY` flag and calls `iderw` to write the buffer to disk.

When the caller is done with a buffer, it must call `brelse` to release it. (The name `brelse`, a shortening of b-release, is cryptic but worth learning: it originated in Unix and is used in BSD, Linux, and Solaris too.) `Brelse` (3974) moves the buffer from its position in the linked list to the front of the list (3981-3986), clears the `B_BUSY` bit, and wakes any processes sleeping on the buffer. Moving the buffer has the effect that the buffers are ordered by how recently they were used (meaning released): the first buffer in the list is the most recently used, and the last is the least recently used. The two loops in `bget` take advantage of this: the scan for an existing buffer must process the entire list in the worst case, but checking the most recently used buffers first (starting at `bcache.head` and following `next` pointers) will reduce scan time when there is good locality of reference. The scan to pick a buffer to reuse picks the least recently used block by scanning backward (following `prev` pointers).

Logging layer

One of the most interesting aspects of file system design is crash recovery. The problem arises because many file system operations involve multiple writes to the disk, and a crash after a subset of the writes may leave the on-disk file system in an inconsistent state. For example, depending on the order of the disk writes, a crash during file deletion may either leave a directory entry pointing to a free inode, or it may leave an allocated but unreferenced inode. The latter is relatively benign, but a directory entry that refers to a freed inode is likely to cause serious problems after a reboot.

Xv6 solves the problem of crashes during file system operations with a simple version of logging. An xv6 system call does not directly write the on-disk file system data structures. Instead, it places a description of all the disk writes it wishes to make in a `log` on the disk. Once the system call has logged its writes, it writes a special `commit` record to the disk indicating the the log contains a complete operation. At that point the system call copies the writes to the on-disk file system data structures. After those writes have completed, the system call erases the log on disk.

If the system should crash and reboot, the file system code recovers from the crash as follows, before running any processes. If the log is marked as containing a complete operation, then the recovery code copies the writes to where they belong in the on-disk file system. If the log is not marked as containing a complete operation, the recovery code ignores it. In either case, the recovery code finishes by erasing the log.

Why does xv6's log solve the problem of crashes during file system operations? If the crash occurs before the operation commits, then the log on disk will not be marked as complete, the recovery code will ignore it, and the state of the disk will be as if the operation had not even started. If the crash occurs after the operation commits, then recovery will replay all of the operation's writes, perhaps repeating them if the operation had started to write them to the on-disk data structure. In either case, the log makes operations atomic with respect to crashes: after recovery, either all of the operation's writes appear on the disk, or none of them appear.

Log design

The log resides at a known fixed location at the very end of the disk. It consists of a header block followed by a sequence of data blocks. The header block contains an array of sector number, one for each of the logged data blocks. The header block also contains the count of logged blocks. Xv6 writes the header block when a transaction commits, but not before, and sets the count to zero after copying the logged blocks to the file system. Thus a crash midway through a transaction will result in a count of zero in the log's header block; a crash after a commit will result in a non-zero count.

Each system call's code indicates the start and end of the sequence of writes that must be atomic; we'll call such a sequence a transaction, though it is much simpler than a database transaction. Only one system call can be in a transaction at any one time: other processes must wait until any ongoing transaction has finished. Thus the

log holds at most one transaction at a time.

Xv6 only allows a single transaction at a time in order to avoid the following kind of race that could occur if concurrent transactions were allowed. Suppose transaction X has written a modification to an inode into the log. Concurrent transaction Y then reads a different inode in the same block, updates that inode, writes the inode block to the log, and commits. It would be a disaster if the commit of Y write X's modified inode to the file system, since X has not yet committed. There are sophisticated ways to solve this problem; xv6 solves it by outlawing concurrent transactions.

Xv6 allows read-only system calls to execute concurrently with a transaction. Inode locks cause the transaction to appear atomic to the read-only system call.

Xv6 dedicates a fixed amount of space on the disk to hold the log. No system call can be allowed to write more distinct blocks than there is space in the log. This is not a problem for most system calls, but two of them can potentially write many blocks: `write` and `unlink`. A large file write may write many data blocks and many bitmap blocks as well as an inode block; unlinking a large file might write many bitmap blocks and an inode. Xv6's write system call breaks up large writes into multiple smaller writes that fit in the log, and `unlink` doesn't cause problems because in practice the xv6 file system uses only one bitmap block.

Code: logging

A typical use of the log in a system call looks like this:

```
begin_trans();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
commit_trans();
```

`begin_trans` (4125) waits until it obtains exclusive use of the log and then returns.

`log_write` (4159) acts as a proxy for `bwrite`; it appends the block's new content to the log and records the block's sector number. `log_write` leaves the modified block in the in-memory buffer cache, so that subsequent reads of the block during the transaction will yield the modified block. `log_write` notices when a block is written multiple times during a single transaction, and overwrites the block's previous copy in the log.

`commit_trans` (4136) first write's the log's header block to disk, so that a crash after this point will cause recovery to re-write the blocks in the log. `commit_trans` then calls `install_trans` (4071) to read each block from the log and write it to the proper place in the file system. Finally `commit_trans` writes the log header with a count of zero, so that a crash after the next transaction starts will result in the recovery code ignoring the log.

`recover_from_log` (4116) is called from `initlog` (4055), which is called during boot before the first user process runs. (2494) It reads the log header, and mimics the actions of `commit_trans` if the header indicates that the log contains a committed transaction.

An example use of the log occurs in `filewrite` (5152). The transaction looks like this:

```
begin_trans();
ilock(f->ip);
r = writei(f->ip, ...);
iunlock(f->ip);
commit_trans();
```

This code is wrapped in a loop that breaks up large writes into individual transactions of just a few sectors at a time, to avoid overflowing the log. The call to `writei` writes many blocks as part of this transaction: the file's inode, one or more bitmap blocks, and some data blocks. The call to `ilock` occurs after the `begin_trans` as part of an overall strategy to avoid deadlock: since there is effectively a lock around each transaction, the deadlock-avoiding lock ordering rule is transaction before inode.

Inodes

The term `inode` can have one of two related meanings. It might refer to the on-disk data structure containing a file's size and list of data block numbers. Or "inode" might refer to an in-memory inode, which contains a copy of the on-disk inode as well as extra information needed within the kernel.

All of the on-disk inodes are packed into a contiguous area of disk called the inode blocks. Every inode is the same size, so it is easy, given a number n, to find the nth inode on the disk. In fact, this number n, called the inode number or i-number, is how inodes are identified in the implementation.

The on-disk inode is defined by a `struct dinode` (3572). The `type` field distinguishes between files, directories, and special files (devices). A type of zero indicates that an on-disk inode is free.

The kernel keeps the set of active inodes in memory; its `struct inode` (3613) is the in-memory copy of a `struct dinode` on disk. The kernel stores an inode in memory only if there are C pointers referring to that inode. The `ref` field counts the number of C pointers referring to the in-memory inode, and the kernel discards the inode from memory if the reference count drops to zero. The `iget` and `iput` functions acquire and release pointers to an inode, modifying the reference count. Pointers to an inode can come from file descriptors, current working directories, and transient kernel code such as `exec`.

The `struct inode` that `iget` returns may not have any useful content. In order to ensure it holds a copy of the on-disk inode, code must call `ilock`. This locks the inode (so that no other process can `ilock` it) and reads the inode from the disk, if it has not already been read. `iunlock` releases the lock on the inode. Separating acquisition of inode pointers from locking helps avoid deadlock in some situations, for example during directory lookup. Multiple processes can hold a C pointer to an inode returned by `iget`, but only one process can lock the inode at a time.

The inode cache only caches inodes to which kernel code or data structures hold C pointers. Its main job is really synchronizing access by multiple processes, not caching. If an inode is used frequently, the buffer cache will probably keep it in mem-

ory if it isn't kept by the inode cache.

Code: Block allocator

Inodes points to blocks that must be allocated. xv6's block allocator maintains a free bitmap on disk, with one bit per block. A zero bit indicates that the corresponding block is free; a one bit indicates that it is in use. The bits corresponding to the boot sector, superblock, inode blocks, and bitmap blocks are always set.

The block allocator provides two functions: `balloc` allocates a new disk block, and `bfree` frees a block. `Balloc` (4304) starts by calling `readsb` to read the superblock from the disk (or buffer cache) into `sb`. `balloc` decides which blocks hold the data block free bitmap by calculating how many blocks are consumed by the boot sector, the superblock, and the inodes (using `BBLOCK`). The loop (4312) considers every block, starting at block 0 up to `sb.size`, the number of blocks in the file system. It looks for a block whose bitmap bit is zero, indicating that it is free. If `balloc` finds such a block, it updates the bitmap and returns the block. For efficiency, the loop is split into two pieces. The outer loop reads each block of bitmap bits. The inner loop checks all `BPB` bits in a single bitmap block. The race that might occur if two processes try to allocate a block at the same time is prevented by the fact that the buffer cache only lets one process use a block at a time.

`Bfree` (4331) finds the right bitmap block and clears the right bit. Again the exclusive use implied by `bread` and `brelse` avoids the need for explicit locking.

Code: Inodes

To allocate a new inode (for example, when creating a file), xv6 calls `ialloc` (4402). `Ialloc` is similar to `balloc`: it loops over the inode structures on the disk, one block at a time, looking for one that is marked free. When it finds one, it claims it by writing the new `type` to the disk and then returns an entry from the inode cache with the tail call to `iget` (4418). Like in `balloc`, the correct operation of `ialloc` depends on the fact that only one process at a time can be holding a reference to `bp`: `ialloc` can be sure that some other process does not simultaneously see that the inode is available and try to claim it.

`Iget` (4453) looks through the inode cache for an active entry (`ip->ref > 0`) with the desired device and inode number. If it finds one, it returns a new reference to that inode. (4462-4466). As `iget` scans, it records the position of the first empty slot (4467-4468), which it uses if it needs to allocate a new cache entry. In both cases, `iget` returns one reference to the caller: it is the caller's responsibility to call `iput` to release the inode. It can be convenient for some callers to arrange to call `iput` multiple times. The function `idup` (4488) increments the reference count so that an additional `iput` call is required before the inode can be dropped from the cache.

Callers must lock the inode using `ilock` before reading or writing its metadata or content. `Ilock` (4502) uses a now-familiar sleep loop to wait for `ip->flag`'s `I_BUSY` bit to be clear and then sets it (4511-4513). Once `ilock` has exclusive access to the inode, it can load the inode metadata from the disk (more likely, the buffer cache) if needed.

The function `iunlock` (4534) clears the `I_BUSY` bit and wakes any processes sleeping in `ilock`.

`Iput` (4552) releases a C pointer to an inode by decrementing the reference count (4568). If this is the last reference, the inode's slot in the inode cache is now free and can be re-used for a different inode.

If `iput` sees that there are no C pointer references to an inode and that the inode has no links to it (occurs in no directory), then the inode and its data blocks must be freed. `Iput` relocks the inode; calls `itrunc` to truncate the file to zero bytes, freeing the data blocks; sets the inode type to 0 (unallocated); writes the change to disk; and finally unlocks the inode (4555-4567).

The locking protocol in `iput` deserves a closer look. The first part worth examining is that when locking `ip`, `iput` simply assumed that it would be unlocked, instead of using a sleep loop. This must be the case, because the caller is required to unlock `ip` before calling `iput`, and the caller has the only reference to it (`ip->ref == 1`). The second part worth examining is that `iput` temporarily releases (4560) and reacquires (4564) the cache lock. This is necessary because `itrunc` and `iupdate` will sleep during disk i/o, but we must consider what might happen while the lock is not held. Specifically, once `iupdate` finishes, the on-disk structure is marked as available for use, and a concurrent call to `ialloc` might find it and reallocate it before `iput` can finish. `Ialloc` will return a reference to the block by calling `iget`, which will find `ip` in the cache, see that its `I_BUSY` flag is set, and sleep. Now the in-core inode is out of sync compared to the disk: `ialloc` reinitialized the disk version but relies on the caller to load it into memory during `ilock`. In order to make sure that this happens, `iput` must clear not only `I_BUSY` but also `I_VALID` before releasing the inode lock. It does this by zeroing `flags` (4565).

Code: Inode contents

The on-disk inode structure, `struct dinode`, contains a size and an array of block numbers (see Figure 5-4). The inode data is found in the blocks listed in the `dinode`'s `addrs` array. The first `NDIRECT` blocks of data are listed in the first `NDIRECT` entries in the array; these blocks are called `direct blocks`. The next `NINDIRECT` blocks of data are listed not in the inode but in a data block called the `indirect block`. The last entry in the `addrs` array gives the address of the indirect block. Thus the first 6 kB (`NDIRECT×BSIZE`) bytes of a file can be loaded from blocks listed in the inode, while the next 64kB (`NINDIRECT×BSIZE`) bytes can only be loaded after consulting the indirect block. This is a good on-disk representation but a complex one for clients. The function `bmap` manages the representation so that higher-level routines such as `readi` and `writei`, which we will see shortly. `Bmap` returns the disk block number of the `bn`'th data block for the inode `ip`. If `ip` does not have such a block yet, `bmap` allocates one.

The function `bmap` (4610) begins by picking off the easy case: the first `NDIRECT` blocks are listed in the inode itself (4615-4619). The next `NINDIRECT` blocks are listed in the indirect block at `ip->addrs[NDIRECT]`. `Bmap` reads the indirect block (4626) and then reads a block number from the right position within the block (4627). If the block

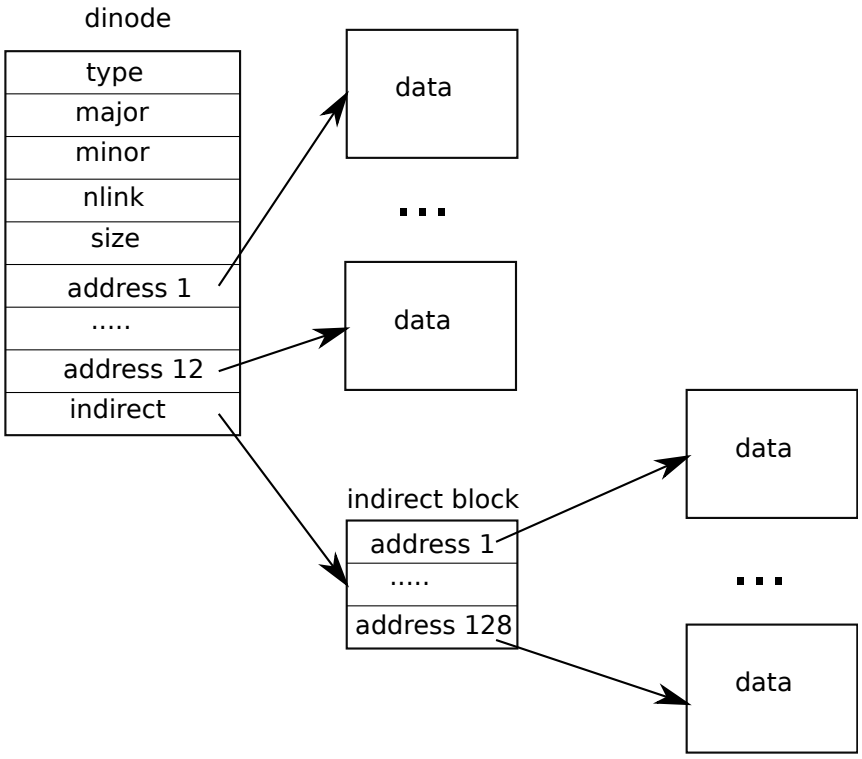

Figure 5-4. The representation of a file on disk.

number exceeds `NDIRECT+NINDIRECT`, `bmap` panics: callers are responsible for not asking about out-of-range block numbers.

`Bmap` allocates block as needed. Unallocated blocks are denoted by a block number of zero. As `bmap` encounters zeros, it replaces them with the numbers of fresh blocks, allocated on demand. (4616-4617, 4624-4625).

`Bmap` allocates blocks on demand as the inode grows; `itrunc` frees them, resetting the inode's size to zero. `Itrunc` (4654) starts by freeing the direct blocks (4660-4665) and then the ones listed in the indirect block (4670-4673), and finally the indirect block itself (4675-4676).

`Bmap` makes it easy to write functions to access the inode's data stream, like `readi` and `writei`. `Readi` (4702) reads data from the inode. It starts making sure that the offset and count are not reading beyond the end of the file. Reads that start beyond the end of the file return an error (4713-4714) while reads that start at or cross the end of the file return fewer bytes than requested (4715-4716). The main loop processes each block of the file, copying data from the buffer into `dst` (4718-4723). The function `writei` (4752) is identical to `readi`, with three exceptions: writes that start at or cross the end of the file grow the file, up to the maximum file size (4765-4766); the loop copies data into the buffers instead of out (4771); and if the write has extended the file, `writei` must update its size (4776-4779).

Both `readi` and `writei` begin by checking for `ip->type == T_DEV`. This case handles special devices whose data does not live in the file system; we will return to this case in the file descriptor layer.

The function stati (4274) copies inode metadata into the stat structure, which is exposed to user programs via the stat system call.

Code: directory layer

The directory layer is simple, because a directory is nothing more that a special kind of file. Its inode has type T_DIR and its data is a sequence of directory entries. Each entry is a struct dirent (3596), which contains a name and an inode number. The name is at most DIRSIZ (14) characters; if shorter, it is terminated by a NUL (0) byte. Directory entries with inode number zero are free.

The function dirlookup (4812) searches a directory for an entry with the given name. If it finds one, it returns a pointer to the corresponding inode, unlocked, and sets *poff to the byte offset of the entry within the directory, in case the caller wishes to edit it. If dirlookup finds an entry with the right name, it updates *poff, releases the block, and returns an unlocked inode obtained via iget. Dirlookup is the reason that iget returns unlocked inodes. The caller has locked dp, so if the lookup was for ., an alias for the current directory, attempting to lock the inode before returning would try to re-lock dp and deadlock. (There are more complicated deadlock scenarios involving multiple processes and .., an alias for the parent directory; . is not the only problem.) The caller can unlock dp and then lock ip, ensuring that it only holds one lock at a time.

The function dirlink (4852) writes a new directory entry with the given name and inode number into the directory dp. If the name already exists, dirlink returns an error (4858-4862). The main loop reads directory entries looking for an unallocated entry. When it finds one, it stops the loop early (4823-4824), with off set to the offset of the available entry. Otherwise, the loop ends with off set to dp->size. Either way, dirlink then adds a new entry to the directory by writing at offset off (4872-4875).

Code: Path names

Like directories, path names require little extra code, because they just call dirlookup recursively, using namei and related functions. Namei (4989) evaluates path as a hierarchical path name and returns the corresponding inode. The function nameiparent is a variant: it stops before the last element, returning the inode of the parent directory and copying the final element into name. Both call the generalized function namex to do the real work.

Namex (4954) starts by deciding where the path evaluation begins. If the path begins with a slash, evaluation begins at the root; otherwise, the current directory (4958-4961). Then it uses skipelem to consider each element of the path in turn (4963). Each iteration of the loop must look up name in the current inode ip. The iteration begins by locking ip and checking that it is a directory. If not, the lookup fails (4964-4968). (Locking ip is necessary not because ip->type can change underfoot—it can't—but because until ilock runs, ip->type is not guaranteed to have been loaded from disk.) If the call is nameiparent and this is the last path element, the loop stops early, as per the definition of nameiparent; the final path element has already been copied into

name, so namex need only return the unlocked ip (4969-4973). Finally, the loop looks for the path element using dirlookup and prepares for the next iteration by setting ip.=.next (4974-4979). When the loop runs out of path elements, it returns ip.

File descriptor layer

One of the cool aspect of the Unix interface is that most resources in Unix are represented as a file, including devices such as the console, pipes, and of course, real files. The file descriptor layer is the layer that achieves this uniformity.

Xv6 gives each process its own table of open files, or file descriptors, as we saw in Chapter 0. Each open file is represented by a struct file (3600), which is a wrapper around either an inode or a pipe, plus an i/o offset. Each call to open creates a new open file (a new struct file): if multiple processes open the same file independently, the different instances will have different i/o offsets. On the other hand, a single open file (the same struct file) can appear multiple times in one process's file table and also in the file tables of multiple processes. This would happen if one process used open to open the file and then created aliases using dup or shared it with a child using fork. A reference count tracks the number of references to a particular open file. A file can be open for reading or writing or both. The readable and writable fields track this.

All the open files in the system are kept in a global file table, the ftable. The file table has a function to allocate a file (filealloc), create a duplicate reference (filedup), release a reference (fileclose), and read and write data (fileread and filewrite).

The first three follow the now-familiar form. Filealloc (5021) scans the file table for an unreferenced file (f->ref == 0) and returns a new reference; filedup (5039) increments the reference count; and fileclose (5052) decrements it. When a file's reference count reaches zero, fileclose releases the underlying pipe or inode, according to the type.

The functions filestat, fileread, and filewrite implement the stat, read, and write operations on files. Filestat (5079) is only allowed on inodes and calls stati. Fileread and filewrite check that the operation is allowed by the open mode and then pass the call through to either the pipe or inode implementation. If the file represents an inode, fileread and filewrite use the i/o offset as the offset for the operation and then advance it (5112-5113, 5165-5166). Pipes have no concept of offset. Recall that the inode functions require the caller to handle locking (5082-5084, 5111-5114, 5164-5178). The inode locking has the convenient side effect that the read and write offsets are updated atomically, so that multiple writing to the same file simultaneously cannot overwrite each other's data, though their writes may end up interlaced.

Code: System calls

With the functions that the lower layers provide the implementation of most system calls is trivial (see sysfile.c). There are a few calls that deserve a closer look.

The functions sys_link and sys_unlink edit directories, creating or removing

references to inodes. They are another good example of the power of using transactions. Sys_link (5313) begins by fetching its arguments, two strings old and new (5318). Assuming old exists and is not a directory (5320-5330), sys_link increments its ip->nlink count. Then sys_link calls nameiparent to find the parent directory and final path element of new (5336) and creates a new directory entry pointing at old's inode (5339). The new parent directory must exist and be on the same device as the existing inode: inode numbers only have a unique meaning on a single disk. If an error like this occurs, sys_link must go back and decrement ip->nlink.

Transactions simplify the implementation because it requires updating multiple disk blocks, but we don't have to worry about the order in which we do them. They either will all succeed or none. For example, without transactions, updating ip->nlink before creating a link, would put the file system temporarily in an unsafe state, and a crash in between could result in havoc. With transactions we don't have to worry about this.

Sys_link creates a new name for an existing inode. The function create (5457) creates a new name for a new inode. It is a generalization of the three file creation system calls: open with the O_CREATE flag makes a new ordinary file, mkdir makes a new directoryy, and mkdev makes a new device file. Like sys_link, create starts by caling nameiparent to get the inode of the parent directory. It then calls dirlookup to check whether the name already exists (5467). If the name does exist, create's behavior depends on which system call it is being used for: open has different semantics from mkdir and mkdev. If create is being used on behalf of open (type == T_FILE) and the name that exists is itself a regular file, then open treats that as a success, so create does too (5471). Otherwise, it is an error (5472-5473). If the name does not already exist, create now allocates a new inode with ialloc (5476). If the new inode is a directory, create initializes it with . and .. entries. Finally, now that the data is initialized properly, create can link it into the parent directory (5489). Create, like sys_link, holds two inode locks simultaneously: ip and dp. There is no possibility of deadlock because the inode ip is freshly allocated: no other process in the system will hold ip's lock and then try to lock dp.

Using create, it is easy to implement sys_open, sys_mkdir, and sys_mknod. Sys_open (5501) is the most complex, because creating a new file is only a small part of what it can do. If open is passed the O_CREATE flag, it calls create (5512). Otherwise, it calls namei (5517). Create returns a locked inode, but namei does not, so sys_open must lock the inode itself. This provides a convenient place to check that directories are only opened for reading, not writing. Assuming the inode was obtained one way or the other, sys_open allocates a file and a file descriptor (5526) and then fills in the file (5534-5538). Note that no other process can access the partially initialized file since it is only in the current process's table.

Chapter 4 examined the implementation of pipes before we even had a file system. The function sys_pipe connects that implementation to the file system by providing a way to create a pipe pair. Its argument is a pointer to space for two integers, where it will record the two new file descriptors. Then it allocates the pipe and installs the file descriptors.

Real world

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

Xv6's logging system is woefully inefficient. It does not allow concurrent updating system calls, even when the system calls operate on entirely different parts of the file system. It logs entire blocks, even if only a few bytes in a block are changed. It performs synchronous log writes, a block at a time, each of which is likely to require an entire disk rotation time. Real logging systems address all of these problems.

Logging is not the only way to provide crash recovery. Early file systems used a scavenger during reboot (for example, the UNIX `fsck` program) to examine every file and directory and the block and inode free lists, looking for and resolving inconsistencies. Scavenging can take hours for large file systems, and there are situations where it is not possible to guess the correct resolution of an inconsistency. Recovery from a log is much faster and is correct.

Xv6 uses the same basic on-disk layout of inodes and directories as early UNIX; this scheme has been remarkably persistent over the years. BSD's UFS/FFS and Linux's ext2/ext3 use essentially the same data structures. The most inefficient part of the file system layout is the directory, which requires a linear scan over all the disk blocks during each lookup. This is reasonable when directories are only a few disk blocks, but is expensive for directories holding many files. Microsoft Windows's NTFS, Mac OS X's HFS, and Solaris's ZFS, just to name a few, implement a directory as an on-disk balanced tree of blocks. This complicated but guarantees logarithmic-time directory lookups.

Xv6 is naive about disk failures: if a disk operation fails, xv6 panics. Whether this is reasonable depends on the hardware: if an operating systems sits atop special hardware that uses redundancy to mask disk failures, perhaps the operating system sees failures so infrequently that panicking is okay. On the other hand, operating systems using plain disks should expect failures and handle them more gracefully, so that the loss of a block in one file doesn't affect the use of the rest of the files system.

Xv6 requires that the file system fit on one disk device and not change in size. As large databases and multimedia files drive storage requirements ever higher, operating systems are developing ways to eliminate the "one disk per file system" bottleneck. The basic approach is to combine many disks into a single logical disk. Hardware solutions such as RAID are still the most popular, but the current trend is moving toward implementing as much of this logic in software as possible. These software implementations typically allowing rich functionality like growing or shrinking the logical device by adding or removing disks on the fly. Of course, a storage layer that can grow or shrink on the fly requires a file system that can do the same: the fixed-size array of inode blocks used by Unix file systems does not work well in such environments. Sepa-

rating disk management from the file system may be the cleanest design, but the complex interface between the two has led some systems, like Sun's ZFS, to combine them.

Xv6's file system lacks many other features in today file systems; for example, it lacks support for snapshots and incremental backup.

Xv6 has two different file implementations: pipes and inodes. Modern Unix systems have many: pipes, network connections, and inodes from many different types of file systems, including network file systems. Instead of the `if` statements in `fileread` and `filewrite`, these systems typically give each open file a table of function pointers, one per operation, and call the function pointer to invoke that inode's implementation of the call. Network file systems and user-level file systems provide functions that turn those calls into network RPCs and wait for the response before returning.

Exercises

1. why panic in balloc? Can we recover?

2. why panic in ialloc? Can we recover?

3. inode generation numbers.

4. Why doesn't filealloc panic when it runs out of files? Why is this more common and therefore worth handling?

5. Suppose the file corresponding to `ip` gets unlinked by another process between `sys_link`'s calls to `iunlock(ip)` and `dirlink`. Will the link be created correctly? Why or why not?

6. `create` makes four function calls (one to `ialloc` and three to `dirlink`) that it requires to succeed. If any doesn't, `create` calls `panic`. Why is this acceptable? Why can't any of those four calls fail?

7. `sys_chdir` calls `iunlock(ip)` before `iput(cp->cwd)`, which might try to lock `cp->cwd`, yet postponing `iunlock(ip)` until after the `iput` would not cause deadlocks. Why not?

Appendix A

PC hardware

This appendix describes personal computer (PC) hardware, the platform on which xv6 runs.

A PC is a computer that adheres to several industry standards, with the goal that a given piece of software can run on PCs sold by multiple vendors. These standards evolve over time and a PC from 1990s doesn't look like a PC now.

From the outside a PC is a box with a keyboard, a screen, and various devices (e.g., CD-rom, etc.). Inside the box is a circuit board (the "motherboard") with CPU chips, memory chips, graphic chips, I/O controller chips, and busses through which the chips communicate. The busses adhere to standard protocols (e.g., PCI and USB) so that devices will work with PCs from multiple vendors.

From our point of view, we can abstract the PC into three components: CPU, memory, and input/output (I/O) devices. The CPU performs computation, the memory contains instructions and data for that computation, and devices allow the CPU to interact with hardware for storage, communication, and other functions.

You can think of main memory as connected to the CPU with a set of wires, or lines, some for address bits, some for data bits, and some for control flags. To read a value from main memory, the CPU sends high or low voltages representing 1 or 0 bits on the address lines and a 1 on the "read" line for a prescribed amount of time and then reads back the value by interpreting the voltages on the data lines. To write a value to main memory, the CPU sends appropriate bits on the address and data lines and a 1 on the "write" line for a prescribed amount of time. Real memory interfaces are more complex than this, but the details are only important if you need to achieve high performance.

Processor and memory

A computer's CPU (central processing unit, or processor) runs a conceptually simple loop: it consults an address in a register called the `program counter`, reads a machine instruction from that address in memory, advances the program counter past the instruction, and executes the instruction. Repeat. If the execution of the instruction does not modify the program counter, this loop will interpret the memory pointed at by the program counter as a sequence of machine instructions to run one after the other. Instructions that do change the program counter include branches and function calls.

The execution engine is useless without the ability to store and modify program data. The fastest storage for data is provided by the processor's register set. A register is a storage cell inside the processor itself, capable of holding a machine word-sized value (typically 16, 32, or 64 bits). Data stored in registers can typically be read or

written quickly, in a single CPU cycle.

PCs have a processor that implements the x86 instruction set, which was originally defined by Intel and has become a standard. Several manufacturers produce processors that implement the instruction set. Like all other PC standards, this standard is also evolving but newer standards are backwards compatible with past standards. The boot loader has to deal with some of this evolution because every PC processor starts simulating an Intel 8088, the CPU chip in the original IBM PC released in 1981. However, for most of xv6 you will be concerned with the modern x86 instruction set.

The modern x86 provides eight general purpose 32-bit registers—`%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%ebp`, and `%esp`—and a program counter `%eip` (the ‘ `instruction pointer`). The common `e` prefix stands for extended, as these are 32-bit extensions of the 16-bit registers `%ax`, `%bx`, `%cx`, `%dx`, `%di`, `%si`, `%bp`, `%sp`, and `%ip`. The two register sets are aliased so that, for example, `%ax` is the bottom half of `%eax`: writing to `%ax` changes the value stored in `%eax` and vice versa. The first four registers also have names for the bottom two 8-bit bytes: `%al` and `%ah` denote the low and high 8 bits of `%ax`; `%bl`, `%bh`, `%cl`, `%ch`, `%dl`, and `%dh` continue the pattern. In addition to these registers, the x86 has eight 80-bit floating-point registers as well as a handful of special-purpose registers like the `control registers` `%cr0`, `%cr2`, `%cr3`, and `%cr4`; the debug registers `%dr0`, `%dr1`, `%dr2`, and `%dr3`; the `segment registers` `%cs`, `%ds`, `%es`, `%fs`, `%gs`, and `%ss`; and the global and local descriptor table pseudo-registers `%gdtr` and `%ldtr`. The control registers and segment registers are important to any operating system. The floating-point and debug registers are less interesting and not used by xv6.

Registers are fast but expensive. Most processors provide at most a few tens of general-purpose registers. The next conceptual level of storage is the main random-access memory (RAM). Main memory is 10-100x slower than a register, but it is much cheaper, so there can be more of it. One reason main memory is relatively slow is that it is physically separate from the processor chip. An x86 processor has a few dozen registers, but a typical PC today has gigabytes of main memory. Because of the enormous differences in both access speed and size between registers and main memory, most processors, including the x86, store copies of recently-accessed sections of main memory in on-chip cache memory. The cache memory serves as a middle ground between registers and memory both in access time and in size. Today's x86 processors typically have two levels of cache, a small first-level cache with access times relatively close to the processor's clock rate and a larger second-level cache with access times in between the first-level cache and main memory. This table shows actual numbers for an Intel Core 2 Duo system:

Intel Core 2 Duo E7200 at 2.53 GHz

TODO: Plug in non-made-up numbers!

storage	**access time**	**size**
register	0.6 ns	64 bytes
L1 cache	0.5 ns	64 kilobytes
L2 cache	10 ns	4 megabytes
main memory	100 ns	4 gigabytes

For the most part, x86 processors hide the cache from the operating system, so we can think of the processor as having just two kinds of storage—registers and memory—and not worry about the distinctions between the different levels of the memory hierarchy.

I/O

Processors must communicate with devices as well as memory. The x86 processor provides special `in` and `out` instructions that read and write values from device addresses called `I/O ports`. The hardware implementation of these instructions is essentially the same as reading and writing memory. Early x86 processors had an extra address line: 0 meant read/write from an I/O port and 1 meant read/write from main memory. Each hardware device monitors these lines for reads and writes to its assigned range of I/O ports. A device's ports let the software configure the device, examine its status, and cause the device to take actions; for example, software can use I/O port reads and writes to cause the disk interface hardware to read and write sectors on the disk.

Many computer architectures have no separate device access instructions. Instead the devices have fixed memory addresses and the processor communicates with the device (at the operating system's behest) by reading and writing values at those addresses. In fact, modern x86 architectures use this technique, called `memory-mapped I/O`, for most high-speed devices such as network, disk, and graphics controllers. For reasons of backwards compatibility, though, the old `in` and `out` instructions linger, as do legacy hardware devices that use them, such as the IDE disk controller, which xv6 uses.

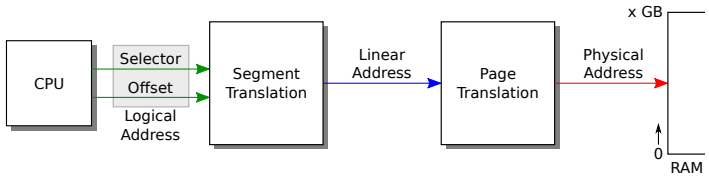

Figure B-1. The relationship between logical, linear, and physical addresses.

Appendix B

The boot loader

When an x86 PC boots, it starts executing a program called the BIOS, which is stored in non-volatile memory on the motherboard. The BIOS's job is to prepare the hardware and then transfer control to the operating system. Specifically, it transfers control to code loaded from the boot sector, the first 512-byte sector of the boot disk. The boot sector contains the boot loader: instructions that load the kernel into memory. The BIOS loads the boot sector at memory address `0x7c00` and then jumps (sets the processor's `%ip`) to that address. When the boot loader begins executing, the processor is simulating an Intel 8088, and the loader's job is to put the processor in a more modern operating mode, to load the xv6 kernel from disk into memory, and then to transfer control to the kernel. The xv6 boot loader comprises two source files, one written in a combination of 16-bit and 32-bit x86 assembly (`bootasm.S`; (8200)) and one written in C (`bootmain.c`; (8300)).

Code: Assembly bootstrap

The first instruction in the boot loader is `cli` (8212), which disables processor interrupts. Interrupts are a way for hardware devices to invoke operating system functions called interrupt handlers. The BIOS is a tiny operating system, and it might have set up its own interrupt handlers as part of the initializing the hardware. But the BIOS isn't running anymore—the boot loader is—so it is no longer appropriate or safe to handle interrupts from hardware devices. When xv6 is ready (in Chapter 2), it will re-enable interrupts.

The processor is in `real mode`, in which it simulates an Intel 8088. In real mode there are eight 16-bit general-purpose registers, but the processor sends 20 bits of address to memory. The segment registers `%cs`, `%ds`, `%es`, and `%ss` provide the additional bits necessary to generate 20-bit memory addresses from 16-bit registers. When a program refers to a memory address, the processor automatically adds 16 times the value of one of the segment registers; these registers are 16 bits wide. Which segment register is usually implicit in the kind of memory reference: instruction fetches use `%cs`, data reads and writes use `%ds`, and stack reads and writes use `%ss`.

Xv6 pretends that an x86 instruction uses a virtual address for its memory operands, but an x86 instruction actually uses a `logical address` (see Figure B-1). A logical address consists of a segment selector and an offset, and is sometimes written as *segment*:*offset*. More often, the segment is implicit and the program only directly manipulates the offset. The segmentation hardware performs the translation described above to generate a `linear address`. If the paging hardware is enabled (see Chapter 1), it translates linear addresses to physical addresses; otherwise the processor uses linear addresses as physical addresses.

The boot loader does not enable the paging hardware; the logical addresses that it uses are translated to linear addresses by the segmentation harware, and then used directly as physical addresses. Xv6 configures the segmentation hardware to translate logical to linear addresses without change, so that they are always equal. For historical reasons we have used the term `virtual address` to refer to addresses manipulated by programs; an xv6 virtual address is the same as an x86 logical address, and is equal to the linear address to which the segmentation hardware maps it. Once paging is enabled, the only interesting address mapping in the system will be linear to physical.

The BIOS does not guarantee anything about the contents of `%ds`, `%es`, `%ss`, so first order of business after disabling interrupts is to set `%ax` to zero and then copy that zero into `%ds`, `%es`, and `%ss` (8215-8218).

A virtual *segment*:*offset* can yield a 21-bit physical address, but the Intel 8088 could only address 20 bits of memory, so it discarded the top bit: `0xffff0+0xffff` = `0x10ffef`, but virtual address `0xffff:0xffff` on the 8088 referred to physical address `0x0ffef`. Some early software relied on the hardware ignoring the 21st address bit, so when Intel introduced processors with more than 20 bits of physical address, IBM provided a compatibility hack that is a requirement for PC-compatible hardware. If the second bit of the keyboard controller's output port is low, the 21st physical address bit is always cleared; if high, the 21st bit acts normally. The boot loader must enable the 21st address bit using I/O to the keyboard controller on ports 0x64 and 0x60 (8220-8236).

Real mode's 16-bit general-purpose and segment registers make it awkward for a program to use more than 65,536 bytes of memory, and impossible to use more than a megabyte. x86 processors since the 80286 have a `protected mode`, which allows physical addresses to have many more bits, and (since the 80386) a "32-bit" mode that causes registers, virtual addresses, and most integer arithmetic to be carried out with 32 bits rather than 16. The xv6 boot sequence enables protected mode and 32-bit mode as follows.

In protected mode, a segment register is an index into a `segment descriptor table` (see Figure B-2). Each table entry specifies a base physical address, a maximum virtual address called the limit, and permission bits for the segment. These permissions are the protection in protected mode: the kernel can use them to ensure that a program uses only its own memory.

xv6 makes almost no use of segments; it uses the paging hardware instead, as Chapter 1 describes. The boot loader sets up the segment descriptor table `gdt` (8282-8285) so that all segments have a base address of zero and the maximum possible limit (four gigabytes). The table has a null entry, one entry for executable code, and one en-

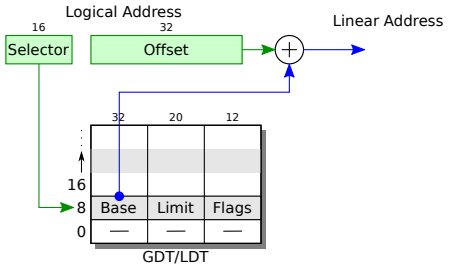

Figure B-2. Segments in protected mode.

try to data. The code segment descriptor has a flag set that indicates that the code should run in 32-bit mode (0660). With this setup, when the boot loader enters protected mode, logical addresses map one-to-one to physical addresses.

The boot loader executes an `lgdt` instruction (8241) to load the processor's global descriptor table (GDT) register with the value `gdtdesc` (8287-8289), which points to the table `gdt`.

Once it has loaded the GDT register, the boot loader enables protected mode by setting the 1 bit (`CR0_PE`) in register `%cr0` (8242-8244). Enabling protected mode does not immediately change how the processor translates logical to physical addresses; it is only when one loads a new value into a segment register that the processor reads the GDT and changes its internal segmentation settings. One cannot directly modify `%cs`, so instead the code executes an `ljmp` (far jump) instruction (8253), which allows a code segment selector to be specified. The jump continues execution at the next line (8256) but in doing so sets `%cs` to refer to the code descriptor entry in `gdt`. That descriptor describes a 32-bit code segment, so the processor switches into 32-bit mode. The boot loader has nursed the processor through an evolution from 8088 through 80286 to 80386.

The boot loader's first action in 32-bit mode is to initialize the data segment registers with `SEG_KDATA` (8258-8261). Logical address now map directly to physical addresses. The only step left before executing C code is to set up a stack in an unused region of memory. The memory from `0xa0000` to `0x100000` is typically littered with device memory regions, and the xv6 kernel expects to be placed at `0x100000`. The boot loader itself is at `0x7c00` through `0x7d00`. Essentially any other section of memory would be a fine location for the stack. The boot loader chooses `0x7c00` (known in this file as `$start`) as the top of the stack; the stack will grow down from there, toward `0x0000`, away from the boot loader.

Finally the boot loader calls the C function `bootmain` (8268). `Bootmain`'s job is to load and run the kernel. It only returns if something has gone wrong. In that case, the code sends a few output words on port `0x8a00` (8270-8276). On real hardware, there is no device connected to that port, so this code does nothing. If the boot loader is running inside a PC simulator, port `0x8a00` is connected to the simulator itself and can transfer control back to the simulator. Simulator or not, the code then executes an infinite loop (8277-8278). A real boot loader might attempt to print an error message first.

Code: C bootstrap

The C part of the boot loader, `bootmain.c` (8300), expects to find a copy of the kernel executable on the disk starting at the second sector. The kernel is an ELF format binary, as we have seen in Chapter 1. To get access to the ELF headers, `bootmain` loads the first 4096 bytes of the ELF binary (8314). It places the in-memory copy at address `0x10000`.

The next step is a quick check that this probably is an ELF binary, and not an uninitialized disk. `Bootmain` reads the section's content starting from the disk location `off` bytes after the start of the ELF header, and writes to memory starting at address `paddr`. `Bootmain` calls `readseg` to load data from disk (8338) and calls `stosb` to zero the remainder of the segment (8340). `Stosb` (0492) uses the x86 instruction `rep stosb` to initialize every byte of a block of memory.

The kernel has been compiled and linked so that it expects to find itself at virtual addresses starting at `0x80100000`. That is, function call instructions mention destination addresses that look like `0xf01xxxxx`; you can see examples in `kernel.asm`. This address is configured in `kernel.ld`. `0x80100000` is a relatively high address, towards the end of the 32-bit address space; Chapter 1 explains the reasons for this choice. There may not be any physical memory at such a high address. Once the kernel starts executing, it will set up the paging hardware to map virtual addresses starting at `0x80100000` to physical addresses starting at `0x00100000`; the kernel assumes that there is physical memory at this lower address. At this point in the boot process, however, paging is not enabled. Instead, `kernel.ld` specifies that the ELF `paddr` start at `0x00100000`, which causes the boot loader to copy the kernel to the low physical addresses to which the paging hardware will eventually point.

The boot loader's final step is to call the kernel's entry point, which is the instruction at which the kernel expects to start executing. For xv6 the entry address is `0x10000c`:

```
# objdump -f kernel

kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

By convention, the `_start` symbol specifies the ELF entry point, which is defined in the file `entry.S` (1036). Since xv6 hasn't set up virtual memory yet, xv6's entry point is the physical address of `entry` (1040).

Real world

The boot loader described in this appendix compiles to around 470 bytes of machine code, depending on the optimizations used when compiling the C code. In order to fit in that small amount of space, the xv6 boot loader makes a major simplifying assumption, that the kernel has been written to the boot disk contiguously starting at sector 1. More commonly, kernels are stored in ordinary file systems, where they may not be contiguous, or are loaded over a network. These complications require the

boot loader to be able to drive a variety of disk and network controllers and understand various file systems and network protocols. In other words, the boot loader itself must be a small operating system. Since such complicated boot loaders certainly won't fit in 512 bytes, most PC operating systems use a two-step boot process. First, a simple boot loader like the one in this appendix loads a full-featured boot-loader from a known disk location, often relying on the less space-constrained BIOS for disk access rather than trying to drive the disk itself. Then the full loader, relieved of the 512-byte limit, can implement the complexity needed to locate, load, and execute the desired kernel. Perhaps a more modern design would have the BIOS directly read a larger boot loader from the disk (and start it in protected and 32-bit mode).

This appendix is written as if the only thing that happens between power on and the execution of the boot loader is that the BIOS loads the boot sector. In fact the BIOS does a huge amount of initialization in order to make the complex hardware of a modern computer look like a traditional standard PC.

Exercises

1. Due to sector granularity, the call to `readseg` in the text is equivalent to `readseg((uchar*)0x100000, 0xb500, 0x1000)`. In practice, this sloppy behavior turns out not to be a problem Why doesn't the sloppy readsect cause problems?
2. something about BIOS lasting longer + security problems
3. Suppose you wanted bootmain() to load the kernel at 0x200000 instead of 0x100000, and you did so by modifying bootmain() to add 0x100000 to the va of each ELF section. Something would go wrong. What?
4. It seems potentially dangerous for the boot loader to copy the ELF header to memory at the arbitrary location `0x10000`. Why doesn't it call `malloc` to obtain the memory it needs?

Index

., 73, 75
.., 73, 75
/init, 28–29
_binary_initcode_size, 25
_binary_initcode_start, 25
_start, 86
acquire, 45, 48
addl, 26
address space, 18
allocproc, 23
allocuvm, 26, 28
alltraps, 34–35
argc, 28
argint, 37
argptr, 37
argstr, 37
argv, 28
atomic, 45
B_BUSY, 39, 65–66
B_DIRTY, 39–41, 65–66
B_VALID, 39–41, 65
balloc, 70
bcache.head, 65
begin_trans, 68–69
bfree, 70
bget, 65
binit, 65
block, 39
bmap, 71
boot loader, 19, 83–85
bootmain, 85
bread, 64, 66
brelse, 64, 66
BSIZE, 71
buf_table_lock, 65
buffer, 39, 64
busy waiting, 40
bwrite, 64, 66, 68
chan, 56, 58
child process, 9
cli, 38, 48
commit, 67
commit_trans, 68
conditional synchronization, 55
contexts, 52
control registers, 80
convoys, 61
copyout, 28
coroutines, 54
cp->killed, 36
cp->tf, 36
cpu->scheduler, 26, 52–53
CR0_PE, 85
CR0_PG, 19
CR0_WP, 19
CR_PSE, 19
crash recovery, 63
create, 75
current directory, 14
deadlocked, 57
direct blocks, 71
dirlink, 73
dirlookup, 73–75
DIRSIZ, 73
DPL_USER, 25, 34
driver, 39
dup, 74
ELF format, 28
ELF_MAGIC, 28
EMBRYO, 24
end, 22–23
enter_alloc, 23
entry, 19, 86
entrypgdir, 19
exception, 31
exec, 9–11, 26–28, 34
exit, 9, 26, 53–54, 60
fetchint, 37
file descriptor, 10
filealloc, 74
fileclose, 74
filedup, 74
fileread, 74, 77
filestat, 74
filewrite, 69, 74, 77
FL, 34
FL_IF, 25
fork, 9–11, 74
forkret, 24, 26, 54
fsck, 76
ftable, 74
gdt, 84–85
gdtdesc, 85
getcmd, 10
global descriptor table, 85
I/O ports, 81
I_BUSY, 70–71
I_VALID, 71
ialloc, 70–71, 75
IDE_BSY, 40
IDE_DRDY, 40
IDE_IRQ, 39
ideinit, 39–40
ideintr, 40, 47–48
idelock, 46, 48
iderw, 40, 46, 48, 65–66
idestart, 40
idewait, 40
idt, 34
idtinit, 38
idup, 70
IF, 38
iget, 69–71, 73
ilock, 69–70, 73
inb, 38
indirect block, 71
initcode, 29
initcode.S, 25–26, 33
initlog, 68
initproc, 26
inituvm, 25
inode, 15, 63, 69
insl, 40
install_trans, 68
instruction pointer, 80
int, 32–34
interface design, 7
interrupt, 31
interrupt handler, 32
ioapicenable, 39
iput, 69–71
iret, 26, 33, 36
IRQ_TIMER,, 38
itrunc, 71–72
iunlock, 71
iupdate, 71
kalloc, 23
KERNBASE, 19

kernel, 7
kernel mode, 32
kernel space, 7
kfree, 22
kinit, 22
kmap, 21
kvmalloc, 20–21
lapicinit, 38
linear address, 83–84
links, 15
loaduvm, 28
lock, 43
log, 67
log_write, 68
logical address, 83–84
main, 20–23, 26, 34, 39, 65
malloc, 10
mappages, 21
memory-mapped I/O, 81
mkdev, 75
mkdir, 75
mpmain, 25
multiplex, 51
namei, 25, 28, 73, 75
nameiparent, 73, 75
namex, 73–74
NBUF, 65
NDIRECT, 71
NINDIRECT, 71
O_CREATE, 75
open, 74–75
outb, 38
p->context, 24, 26, 54
p->cwd, 25
p->kstack, 23, 60
p->name, 25
p->parent, 60
p->pgdir, 23, 60
p->state, 23
p->sz, 37
p->xxx, 23
page, 17
page directory, 17
page table entries (PTEs), 17
page table pages, 17
panic, 36
parent process, 9
path, 14
persistence, 63
PGROUNDUP, 23
physical address, 17, 83
PHYSTOP, 21–22
picenable, 39
pid, 9, 24
pipe, 13
piperead, 59
pipewrite, 59
polling, 40
popal, 26
popcli, 48
popl, 26
printf, 9
priority inversion, 61
process, 7–8, 17
program counter, 79
programmable interrupt controler (PIC), 38
protected mode, 84–85
ptable, 47
ptable.lock, 53–54, 58, 60
PTE_P, 18
PTE_U, 18, 21, 26
PTE_W, 18
pushcli, 48
race condition, 44
read, 74
readi, 28, 71–72
readsb, 70
readseg, 86
real mode, 83
recover_from_log, 68
recursive locks, 46
release, 46, 48–49
ret, 26
root, 14
round robin, 60
RUNNABLE, 25, 54, 58–59
sbrk, 10, 27
sched, 52–54, 58, 60
scheduler, 25–26, 53–54
sector, 39
SEG_KCPU, 35
SEG_KDATA, 26, 85
SEG_TSS, 26
SEG_UCODE, 25
SEG_UDATA, 25
seginit, 29
segment descriptor table, 84
segment registers, 80
sequence coordination, 55
setupkvm, 21, 23, 25–26, 28
skipelem, 73
sleep, 47, 53, 56–58, 65
SLEEPING, 58
stat, 73–74
stati, 73–74
sti, 38, 48
stosb, 86
struct buf, 39
struct context, 52
struct dinode, 69, 71
struct dirent, 73
struct elfhdr, 28
struct file, 74
struct inode, 69
struct pipe, 59
struct proc, 23, 60
struct run, 22
struct spinlock, 45
struct trapframe, 25
superblock, 63
superpage, 19
switch, 26
switchuvm, 26, 34, 38, 54
swtch, 26, 52–54, 60
sys_exec, 27, 34
SYS_exec, 26, 36
sys_link, 74–75
sys_mkdir, 75
sys_mknod, 75
sys_open, 75
sys_pipe, 75
sys_sleep, 48
sys_unlink, 74
syscall, 27, 36
system calls, 7
T_DEV, 72
T_DIR, 73
T_FILE, 75
T_SYSCALL, 26, 34, 36
tf->trapno, 36
thread, 23
thundering herd, 61
ticks, 48
tickslock, 48
timer.c, 38
transaction, 63
trap, 35–36, 39–40, 52
trapret, 24, 26, 36
traps, 32

tvinit, 34
type cast, 23
unlink, 68
user memory, 18
user mode, 32
user space, 7
userinit, 23, 25–26
ustack, 28
V2P_WO, 19
vectors[i], 34
virtual address, 17, 84
wait channel, 56
wait, 9, 54, 60
wakeup, 39, 47, 56, 58
wakeup1, 58
walkpgdir, 21–23, 28
write, 68, 74
writei, 69, 71–72
xchg, 45, 48
yield, 52–54
ZOMBIE, 60