# xv6

a simple, Unix-like teaching operating system

Russ Cox Frans Kaashoek Robert Morris

[xv6-book@pdos.csail.mit.edu](mailto:xv6-book@pdos.csail.mit.edu)

Draft as of September 4, 2018

|  |  |  |
| --- | --- | --- |
| **Contents** |  | |
| 0 | [Operating system interfaces](#_bookmark0) | *7* |
| 1 | [Operating system organization](#_bookmark3) | *17* |
| 2 | [Page tables](#_bookmark7) | *29* |
| 3 | [Traps, interrupts, and drivers](#_bookmark11) | *39* |
| 4 | [Locking](#_bookmark14) | *51* |
| 5 | [Scheduling](#_bookmark17) | *61* |
| 6 | [File system](#_bookmark20) | *75* |
| 7 | [Summary](#_bookmark24) | *93* |
| A | [PC hardware](#_bookmark25) | *95* |
| B | [The boot loader](#_bookmark27) | *99* |
| [Index](#_bookmark30) | | *105* |

# Foreword and acknowledgements

This is a draft text intended for a class on operating systems. It explains the main con- cepts of operating systems by studying an example kernel, named xv6. xv6 is a re-im- plementation of Dennis Ritchie’s and Ken Thompson’s Unix Version 6 (v6). xv6 loose- ly 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; IS- BN: 1-57398-013-7; 1st edition (June 14, 2000)). See https://pdos.csail.mit.edu/6.828 for pointers to on-line resources for v6 and xv6, including several hands-on homework as- signments using xv6.

We have used this text in 6.828, the operating systems class at MIT. We thank the fac- ulty, teaching assistants, and students of 6.828 who have all directly or indirectly con- tributed to xv6. In particular, we would like to thank Austin Clements and Nickolai Zeldovich. Finally, we would like to thank people who emailed us bugs in the text or suggestions for provements: Abutalib Aghayev, Sebastian Boehm, Anton Burtsev, Raphael Carvalho, Rasit Eskicioglu, Color Fuzzy, Giuseppe, Tao Guo, Robert Hilder- man, Wolfgang Keller, Austin Liew, Pavan Maddamsetti, Jacek Masiulaniec, Michael McConville, miguelgvieira, Mark Morrissey, Harry Pan, Askar Safin, Salman Shah, Rus- lan Savchenko, Pawel Szczurko, Warren Toomey, tyfkda, and Zou Chang Wei.

If you spot errors or have suggestions for improvement, please send email to Frans Kaashoek and Robert Morris (kaashoek,rtm@csail.mit.edu).

# 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 and abstracts the low-level hardware, so that, for example, a word processor need not concern itself with which type of disk hardware is being used. It also shares the hardware among multiple programs so that they run (or ap- pear 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 an 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 imple- mentation 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 oper- ating system concepts. That operating system, xv6, provides the basic interfaces intro- duced by Ken Thompson and Dennis Ritchie’s Unix operating system, as well as mim- icking 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](#_bookmark1), xv6 takes the traditional form of a *kernel*, a special pro- gram that provides services to running programs. Each running program, called a *pro- cess*, has memory containing instructions, data, and a stack. The instructions imple- ment the program’s computation. The data are the variables on which the computa- tion 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 a procedure is called a *system call*. The system call enters the kernel; the kernel performs the service and returns. Thus a process al- ternates 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.

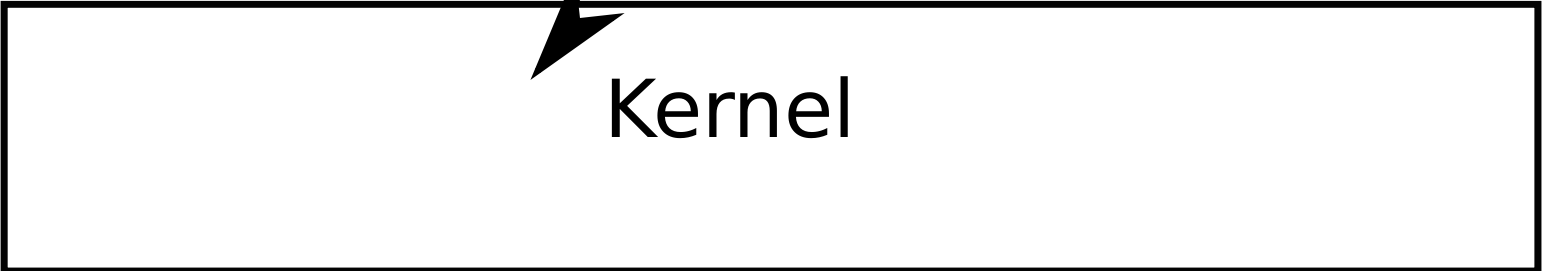
The collection of system calls that a kernel provides is the interface that user pro- grams see. The xv6 kernel provides a subset of the services and system calls that Unix kernels traditionally offer. [Figure 0-2](#_bookmark2) lists all of xv6’s system calls.

interface design kernel

process system call user space kernel space

process

Kernel



user space

shell

cat

system call

kernel space



**Figure 0-1**. A kernel and two user processes.

The rest of this chapter outlines xv6’s services—processes, memory, file descrip- tors, pipes, and file system—and illustrates them with code snippets and discussions of how the *shell*, which is the primary user interface to traditional Unix-like systems, 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 exe- cutes them. 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; as a result, modern Unix systems have a variety of shells to choose from, each with its own user interface and scripting features. The xv6 shell is a simple implementation of the essence of the Unix Bourne shell. Its im-

plementation can be found at line (8550).

## Processes and memory

An xv6 process consists of user-space memory (instructions, data, and stack) and per-process state private to the kernel. Xv6 can *time-share* processes: 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. The kernel associates a process identifier, or pid, with each process.

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 = 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");

shell

time-share pid+code fork+code child process parent process fork+code exit+code wait+code

}

The exit system call causes the calling process to stop executing and to release re- sources such as memory and open files. The wait system call returns the pid of an

**System call Description**

fork() Create a process

exit() Terminate the current process

wait() Wait for a child process to exit

kill(pid) Terminate process pid

getpid() Return the current process’s pid

sleep(n) Sleep for n clock ticks exec(filename, \*argv) Load a file and execute it

sbrk(n) Grow process’s memory by n bytes open(filename, flags) Open a file; the flags indicate read/write read(fd, buf, n) Read n bytes 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

**Figure 0-2**. Xv6 system calls

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 the child exits the parent’s wait returns, causing the parent to print

wait+code printf+code wait+code exec+code exec+code

parent: child 1234 is done

Although the child has the same memory contents as the parent initially, the parent and child are executing with different memory and different registers: changing a vari- able in one does not affect the other. For example, when the return value of wait is stored into pid in the parent process, it doesn’t change the variable pid in the child. The value of pid in the child will still be zero.

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 for- mat, which specifies which part of the file holds instructions, which part is data, at which instruction to start, etc. xv6 uses the ELF format, which Chapter 2 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 (8701). The main loop reads a line of input from the user with getcmd. Then it calls fork, which creates a copy of the shell pro- cess. The parent calls wait, while the child runs the command. For example, if the user had typed ‘‘echo hello’’ to the shell, runcmd would have been called with ‘‘echo hello’’ as the argument. runcmd (8606) runs the actual command. For ‘‘echo hello’’, it would call exec (8626). 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 par- ent to return from wait in main (8701). 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 re- quired 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.

## I/O and 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 descrip- tor. For simplicity we’ll often refer to the object a file descriptor refers to as a ‘‘file’’; the file descriptor interface abstracts away the differences between files, pipes, and de- vices, making them all look like streams of bytes.

Internally, the xv6 kernel uses the file descriptor as an index into a per-process ta- ble, 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 (8707), 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

getcmd+code fork+code exec+code fork+code exec+code malloc+code sbrk+code file descriptor

file descriptor fd, copies them into buf, and returns the number of bytes read. Each file descriptor that refers to a file 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 er- ror 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();

fork+code exec+code

}

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 conven- tion that file descriptor 0 is input and file descriptor 1 is output allows a simple imple- mentation of cat.

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

File descriptors and fork interact to make I/O redirection easy to implement. Fork copies the 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 exec- ing 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 in exactly this way (8630). Re- call 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. Because if they are separate, the shell can fork a child, use open, close, dup in the child to change the standard input and output file descriptors, and then exec. No changes to the program being exec-ed (cat in our example) are required. If fork and exec were combined into a single system call, some other (probably more complex) scheme would be required for the shell to redirect standard input and output, or the program itself would have to understand how to redirect I/O.

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 sequential output 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 dupli- cate 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 redi- rection for the error file descriptor, but now you know how to 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.

## 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 { close(p[0]);

write(p[1], "hello world\n", 12); close(p[1]);

pipe

}

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 refer- ring to the pipe. The child dups the read end onto file descriptor 0, closes the file de- scriptors in p, and execs wc. When wc reads from its standard input, it reads from the pipe. The parent closes the read side of the pipe, writes to the pipe, and then closes the write side.

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 re- turn 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 pipelines such as grep fork sh.c | wc -l in a man- ner similar to the above code (8650). The child process creates a pipe to connect the left end of the pipeline with the right end. Then it calls fork and runcmd for the left end of the pipeline and fork and runcmd for the right end, and waits for both to fin- ish. The right end of the pipeline 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 pipeline, but doing so correctly would complicate the implementation.

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

echo hello world | wc

could be implemented without pipes as

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

Pipes have at least four advantages over temporary files in this situation. 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 parallel execution of pipeline stages, while the file ap- proach requires the first program to finish before the second starts. Fourth, if you are implementing inter-process communication, pipes’ blocking reads and writes are more efficient than the non-blocking semantics of files.

## File system

The xv6 file system provides data files, which are uninterpreted byte arrays, and directories, which contain named references to data files and other directories. The di- rectories form 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 directo- ry 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 (assuming all the directories in- volved exist):

chdir("/a");

chdir("b");

open("c", O\_RDONLY);

open("/a/b/c", O\_RDONLY);

The first fragment changes the process’s current directory to /a/b; the second neither refers to nor changes 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:

root path

current directory

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 pro- cess 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 // Device

struct stat {

short type; // Type of file

int dev; // File system’s disk device uint ino; // Inode number

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 *in- ode*, 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. Further- more,

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 pro- cess closes fd or exits.

Shell commands for file system operations are implemented as user-level pro- grams such as mkdir, ln, rm, etc. This design allows anyone to extend the shell with

inode links

new user commands by just adding a new user-level program. In hindsight this plan seems obvious, but other systems designed at the time of Unix often built such com- mands into the shell (and built the shell into the kernel).

One exception is cd, which is built into the shell (8716). cd must change the cur- rent 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, and cd would change the *child*’s working directory. 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 re- sponsible 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.

The Unix system call interface has been standardized through the Portable Oper- ating System Interface (POSIX) standard. Xv6 is *not* POSIX compliant. It misses sys- tem calls (including basic ones such as lseek), it implements systems calls only par- tially, etc. Our main goals for xv6 are simplicity and clarity while providing a simple UNIX-like system-call interface. Several people have extended xv6 with a few more basic system calls and a simple C library so that they can run basic Unix programs. Modern kernels, however, provide many more system calls, and many more kinds of kernel services, than xv6. For example, they support networking, windowing systems, user-level threads, drivers for many devices, and so on. Modern kernels evolve contin- uously and rapidly, and offer many features beyond POSIX.

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 dis- cussed above. The authors of Unix went on to build Plan 9, which applied the ‘‘re- sources are files’’ concept to modern facilities, representing networks, graphics, and oth- er resources as files or file trees.

The file system abstraction has been a powerful idea. Even so, there are other models for operating system interfaces. Multics, a predecessor of Unix, abstracted file storage in a way that made it look like memory, 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 process- es onto the underlying hardware, isolate processes from each other, and provide mech- anisms 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

**Operating system organization**

A key requirement for an operating system is to support several activities at once. For example, using the system call interface described in chapter 0 a process can start new processes with fork. The operating system must *time-share* the resources of the computer among these processes. For example, even if there are more processes than there are hardware processors, the operating system must ensure that all of the pro- cesses make progress. The operating system must also arrange for *isolation* between the processes. That is, if one process has a bug and fails, it shouldn’t affect processes that don’t depend on the failed process. Complete isolation, however, is too strong, since it should be possible for processes to interact; pipelines are an example. Thus an operating system must fulfil three requirements: multiplexing, isolation, and interaction.

This chapter provides an overview of how operating systems are organized to achieve these 3 requirements. It turns out there are many ways to do so, but this text focuses on mainstream designs centered around a *monolithic kernel*, which is used by many Unix operating systems. This chapter introduces xv6’s design by tracing the cre- ation of the first process when xv6 starts running. In doing so, the text provides a glimpse of the implementation of all major abstractions that xv6 provides, how they interact, and how the three requirements of multiplexing, isolation, and interaction are met. Most of xv6 avoids special-casing the first process, and instead reuses code that xv6 must provide for standard operation. Subsequent chapters will explore each ab- straction in more detail.

Xv6 runs on Intel 80386 or later (‘‘x86’’) processors on a PC platform, and much of its low-level functionality (for example, its process 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.

## Abstracting physical resources

The first question one might ask when encountering an operating system is why have it at all? That is, one could implement the system calls in [Figure 0-2](#_bookmark2) as a library, with which applications link. In this plan, each application could even have its own li- brary tailored to its needs. Applications could directly interact with hardware re- sources and use those resources in the best way for the application (e.g., to achieve high or predictable performance). Some operating systems for embedded devices or real-time systems are organized in this way.

The downside of this library approach is that, if there is more than one applica-

time-share isolation monolithic kernel

tion running, the applications must be well-behaved. For example, each application must periodically give up the processor so that other applications can run. Such a *co- operative* time-sharing scheme may be OK if all applications trust each other and have no bugs. It’s more typical for applications to not trust each other, and to have bugs, so one often wants stronger isolation than a cooperative scheme provides.

To achieve strong isolation it’s helpful to forbid applications from directly access- ing sensitive hardware resources, and instead to abstract the resources into services. For example, applications interact with a file system only through open, read, write, and close system calls, instead of read and writing raw disk sectors. This provides the application with the convenience of pathnames, and it allows the operating system (as the implementor of the interface) to manage the disk.

Similarly, Unix transparently switches hardware processors among processes, sav- ing and restoring register state as necessary, so that applications don’t have to be aware of time sharing. This transparency allows the operating system to share processors even if some applications are in infinite loops.

As another example, Unix processes use exec to build up their memory image, instead of directly interacting with physical memory. This allows the operating system to decide where to place a process in memory; if memory is tight, the operating sys- tem might even store some of a process’s data on disk. Exec also provides users with the convenience of a file system to store executable program images.

Many forms of interaction among Unix processes occur via file descriptors. Not only do file descriptors abstract away many details (e.g. where data in a pipe or file is stored), they also are defined in a way that simplifies interaction. For example, if one application in a pipeline fails, the kernel generates end-of-file for the next process in the pipeline.

As you can see, the system call interface in [Figure 0-2](#_bookmark2) is carefully designed to provide both programmer convenience and the possibility of strong isolation. The Unix interface is not the only way to abstract resources, but it has proven to be a very good one.

## User mode, kernel mode, and system calls

Strong isolation requires a hard boundary between applications and the operating system. If the application makes a mistake, we don’t want the operating system to fail or other applications to fail. Instead, the operating system should be able to clean up the failed application and continue running other applications. To achieve strong iso- lation, the operating system must arrange that applications cannot modify (or even read) the operating system’s data structures and instructions and that applications can- not access other process’s memory.

Processors provide hardware support for strong isolation. For example, the x86 processor, like many other processors, has two modes in which the processor can exe- cute instructions: *kernel mode* and *user mode*. In kernel mode the processor is allowed to execute *privileged instructions*. For example, reading and writing the disk (or any other I/O device) involves privileged instructions. If an application in user mode at-

kernel mode user mode privileged

instructions

tempts to execute a privileged instruction, then the processor doesn’t execute the in- struction, but switches to kernel mode so that the software in kernel mode can clean up the application, because it did something it shouldn’t be doing. [Figure 0-1](#_bookmark1) in Chap- ter 0 illustrates this organization. An application can execute only user-mode instruc- tions (e.g., adding numbers, etc.) and is said to be running in *user space*, while the software in kernel mode can also execute privileged instructions and is said to be run- ning in *kernel space*. The software running in kernel space (or in kernel mode) is called the *kernel*.

An application that wants to read or write a file on disk must transition to the

kernel to do so, because the application itself can not execute I/O instructions. Proces- sors provide a special instruction that switches the processor from user mode to kernel mode and enters the kernel at an entry point specified by the kernel. (The x86 proces- sor provides the int instruction for this purpose.) Once the processor has switched to kernel mode, the kernel can then validate the arguments of the system call, decide whether the application is allowed to perform the requested operation, and then deny it or execute it. It is important that the kernel sets the entry point for transitions to kernel mode; if the application could decide the kernel entry point, a malicious appli- cation could enter the kernel at a point where the validation of arguments etc. is skipped.

## Kernel organization

A key design question is what part of the operating system should run in kernel mode. One possibility is that the entire operating system resides in the kernel, so that the implementations of all system calls run in kernel mode. This organization is called a *monolithic kernel*.

In this organization the entire operating system runs with full hardware privilege. This organization is convenient because the OS designer doesn’t have to decide which part of the operating system doesn’t need full hardware privilege. Furthermore, it easy for different parts of the operating system to cooperate. For example, an operating system might have a buffer cache that can be shared both by the file system and the virtual memory system.

A downside of the monolithic organization is that the interfaces between different parts of the operating system are often complex (as we will see in the rest of this text), and therefore it is easy for an operating system developer to make a mistake. In a monolithic kernel, a mistake is fatal, because an error in kernel mode will often result in the kernel to fail. If the kernel fails, the computer stops working, and thus all appli- cations fail too. The computer must reboot to start again.

To reduce the risk of mistakes in the kernel, OS designers can minimize the amount of operating system code that runs in kernel mode, and execute the bulk of the operating system in user mode. This kernel organization is called a *microkernel*.

[Figure 1-1](#_bookmark4) illustrates this microkernel design. In the figure, the file system runs as a user-level process. OS services running as processes are called servers. To allow ap- plications to interact with the file server, the kernel provides an inter-process commu- nication mechanism to send messages from one user-mode process to another. For

user space kernel space kernel

monolithic kernel microkernel

microkernel

Microkernel



user space

shell File server

Send message

kernel space



**Figure 1-1**. A microkernel with a file system server

example, if an application like the shell wants to read or write a file, it sends a message to the file server and waits for a response.

In a microkernel, the kernel interface consists of a few low-level functions for starting applications, sending messages, accessing device hardware, etc. This organiza- tion allows the kernel to be relatively simple, as most of the operating system resides in user-level servers.

Xv6 is implemented as a monolithic kernel, following most Unix operating sys- tems. Thus, in xv6, the kernel interface corresponds to the operating system interface, and the kernel implements the complete operating system. Since xv6 doesn’t provide many services, its kernel is smaller than some microkernels.

## Process overview

The unit of isolation in xv6 (as in other Unix operating systems) is a *process*. The process abstraction prevents one process from wrecking or spying on another process’s memory, CPU, file descriptors, etc. It also prevents a process from wrecking the kernel itself, so that a process can’t subvert the kernel’s isolation mechanisms. The kernel must implement the process abstraction with care because a buggy or malicious appli- cation may trick the kernel or hardware in doing something bad (e.g., circumventing enforced isolation). The mechanisms used by the kernel to implement processes in- clude the user/kernel mode flag, address spaces, and time-slicing of threads.

To help enforce isolation, the process abstraction provides the illusion to a pro- gram that it has its own private machine. A process provides a program with what ap- pears to be a private memory system, or *address space*, which other processes cannot read or write. A process also provides the program with what appears to be its own CPU to execute the program’s instructions.

Xv6 uses page tables (which are implemented by hardware) to give each process its own address space. The x86 page table translates (or ‘‘maps’’) a *virtual address* (the address that an x86 instruction manipulates) to a *physical address* (an address that the processor chip sends to main memory).

Xv6 maintains a separate page table for each process that defines that process’s address space. As illustrated in [Figure 1-2](#_bookmark5), an address space includes the process’s *user memory* starting at virtual address zero. Instructions come first, followed by global variables, then the stack, and finally a ‘‘heap’’ area (for malloc) that the process can ex- pand as needed.

process address space virtual address

physical address user memory

0xFFFFFFFF

|  |
| --- |
|  |
| free memory |
| text and data |
| BIOS |
| heap |
| user stack |
| user text and data |

0x80100000



0x80000000

kernel

user

0

**Figure 1-2**. Layout of a virtual address space

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 plenty of room for user memory, xv6’s address spaces map the kernel at high ad- dresses, starting at 0x80100000.

The xv6 kernel maintains many pieces of state for each process, which it gathers into a struct proc (2337). A process’s most important pieces of kernel state are its page table, its kernel stack, and its run state. We’ll use the notation p->xxx to refer to elements of the proc structure.

Each process has a thread of execution (or *thread* for short) that executes the pro- cess’s instructions. A thread can be suspended and later resumed. To switch transpar- ently between processes, the kernel suspends the currently running thread and resumes another process’s thread. Much of the state of a thread (local variables, function call return addresses) is stored on the thread’s stacks. Each process has two stacks: a user stack and a kernel stack (p->kstack). When the process is executing user instructions, only its user stack is in use, and its kernel stack is empty. When the process enters the kernel (for a system call or interrupt), the kernel code executes on the process’s kernel stack; while a process is in the kernel, its user stack still contains saved data, but isn’t actively used. A process’s thread alternates between actively using its user stack and its kernel stack. The kernel stack is separate (and protected from user code) so that the kernel can execute even if a process has wrecked its user stack.

When a process makes a system call, the processor switches to the kernel stack, raises the hardware privilege level, and starts executing the kernel instructions that im- plement the system call. When the system call completes, the kernel returns to user space: the hardware lowers its privilege level, switches back to the user stack, and re- sumes executing user instructions just after the system call instruction. A process’s thread can ‘‘block’’ in the kernel to wait for I/O, and resume where it left off when the

struct proc+code p->xxx+code thread

p->kstack+code

0xFFFFFFFF



BIOS

kernel text and data

text and data

BIOS

text and data

0x80100000

0x80000000

Top physica memory

4 Mbyte

0 0

Virtual address space Physical memory

**Figure 1-3**. Layout of a virtual address space

I/O has finished.

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

p->pgdir holds the process’s page table, in the format that the x86 hardware ex- pects. 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.

## Code: the first address space

To make the xv6 organization more concrete, we’ll look how the kernel creates the first address space (for itself), how the kernel creates and starts the first process, and how that process performs the first system call. By tracing these operations we see in detail how xv6 provides strong isolation for processes. The first step in providing strong iso- lation is setting up the kernel to run in its own address space.

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 (1044). The x86 paging hard- ware 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 I/O devices.

To allow the rest of the kernel to run, entry sets up a page table that maps virtu-

p->state+code p->pgdir+code boot loader entry+code

al addresses starting at 0x80000000 (called KERNBASE (0207)) to physical addresses start- ing at 0x0 (see [Figure 1-2](#_bookmark5)). Setting up two ranges of virtual addresses that map to the same physical memory range is a common use of page tables, and we will see more examples like this one.

The entry page table is defined in main.c (1306). We look at the details of page ta- bles in Chapter 2, but the short story is that 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.

Entry 512 maps virtual addresses KERNBASE:KERNBASE+0x400000 to physical ad- dresses 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, it loads the physical address of entrypgdir into control reg- ister %cr3. The value in %cr3 must be a physical address. It wouldn’t make sense for

%cr3 to hold the virtual address of entrypgdir, because the paging hardware doesn’t know how to translate virtual addresses yet; it doesn’t have a page table yet. The sym- bol entrypgdir refers to an address in high memory, and the macro V2P\_WO (0213) subtracts KERNBASE in order to find the physical address. To enable the paging hard- ware, xv6 sets the flag CR0\_PG in the control register %cr0.

The processor is still executing instructions at low addresses after paging is en- abled, 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- 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 makes the stack pointer, %esp, point to memory to be used as a stack (1058). 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 otherwise 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 (1217).

## Code: creating the first process

Now we’ll look at how the kernel creates user-level processes and ensures that they are strongly isolated.

After main (1217) initializes several devices and subsystems, it creates the first pro- cess by calling userinit (2520). Userinit’s first action is to call allocproc. The job of allocproc (2473) is 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. Al- locproc is called for each new process, while userinit is called only for the very first process. Allocproc scans the proc table for a slot with state UNUSED (2480-2482). When it finds an unused slot, allocproc sets the state to EMBRYO to mark it as used and gives the process a unique pid (2469-2489). Next, it tries to allocate a kernel stack for

KERNBASE+code entry+code entrypgdir+code V2P\_WO+code CR0\_PG+code main+code main+code main+code allocproc+code EMBRYO+code pid+code

top of new stack

|  |
| --- |
| esp |
| ... |
| eip |
| ... |
| edi |
| trapret |
| eip |
| ... |
| edi |
| (empty) |

p->tf address forkret will return to

p->context

**Figure 1-4**. A new kernel stack.

p->kstack

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. allocproc is written so that it can be used by fork as well as when creating the first process. allocproc sets up the new process with a specially prepared kernel stack and set of kernel regis- ters that cause it to ‘‘return’’ to user space when it first runs. The layout of the pre- pared kernel stack will be as shown in [Figure 1-4](#_bookmark6). 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 (2507-2512). 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 (2853). This function will return to whatever address is at the bottom of the stack. The context switch code (3059) 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 regis- ters from values stored at the top of the kernel stack and jumps into the process (3324). 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 user-space location zero rather than at a return from fork.

As we will see in Chapter 3, 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 process’s kernel stack. userinit writes values at the top of the new stack that look just like those that would

forkret+code trapret+code

p->context+code forkret+code trapret+code forkret+code trapret+code userinit+code

be there if the process had entered the kernel via an interrupt (2533-2539), so that the or- dinary 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 1-4](#_bookmark6).

The first process is going to execute a small program (initcode.S; (8400)). The

struct

trapframe+code initcode.S+code userinit+code setupkvm+code initcode.S+code

\_binary\_initcode\_start

process needs physical memory in which to store this program, the program needs to

\_binary\_initcode\_size+

be copied to that memory, and the process needs a page table that maps user-space addresses to that memory.

userinit calls setupkvm (1818) to create a page table for the process with (at first) mappings only for memory that the kernel uses. We will study this function in detail in Chapter 2, but at a high level setupkvm and userinit create an address space as shown in [Figure 1-2](#_bookmark5).

The initial contents of the first process’s user-space 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 \_bina- ry\_initcode\_size, indicating 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 bi- nary to that page (1886).

Then userinit sets up the trap frame (0602) with the initial user mode state: the

%cs register contains a segment selector for the SEG\_UCODE segment running at privi- lege level DPL\_USER (i.e., user mode rather than kernel mode), and similarly %ds, %es, and %ss use SEG\_UDATA with privilege DPL\_USER. The %eflags FL\_IF bit is set to al- low hardware interrupts; we will reexamine this in Chapter 3.

The stack pointer %esp is set to the process’s largest valid virtual address, p->sz.

The instruction pointer is set to 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 6.

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

## Code: Running the first 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 (1257). Scheduler (2758) looks for a process with p->state set to RUNNABLE, and there’s only one: initproc. It sets the per-cpu variable proc to the process it found and calls switchuvm to tell the hardware to start using the target process’s page table (1879). 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 sets up a task state segment SEG\_TSS that instructs the hardware to execute system calls and inter- rupts on the process’s kernel stack. We will re-examine the task state segment in Chapter 3.

scheduler now sets p->state to RUNNING and calls swtch (3059) to perform a

inituvm+code

SEG\_UCODE+code DPL\_USER+code SEG\_UDATA+code DPL\_USER+code FL\_IF+code userinit+code

p->name+code p->cwd+code namei+code userinit+code

RUNNABLE+code mpmain+code scheduler+code switchuvm+code setupkvm+code SEG\_TSS+code scheduler+code swtch+code

context switch to the target process’s kernel thread. swtch first saves the current regis- ters. The current context is not a process but rather a special per-cpu scheduler con- text, so scheduler tells swtch to save the current hardware registers in per-cpu stor- age (cpu->scheduler) rather than in any process’s kernel thread context. swtch then loads the saved registers of the target kernel thread (p->context) into the x86 hard- ware registers, including the stack pointer and instruction pointer. We’ll examine swtch in more detail in Chapter 5. The final ret instruction (3078) pops the target process’s %eip from the stack, finishing the context switch. Now the processor is run- ning on the kernel stack of process p.

Allocproc had previously set initproc’s p->context->eip to forkret, so the ret starts executing forkret. On the first invocation (that is this one), forkret (2853) 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 (3324) uses pop instructions to restore registers from the trap frame (0602) 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 instruction pops %cs, %eip, %flags, %esp, and

%ss from 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 init- proc, 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 has set up the process’s page table so that virtual address zero refers to the physical memory allocated for this process, and set a flag (PTE\_U) that tells the paging hardware to allow user code to access that memory. The fact that userinit (2533) 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 pages with PTE\_U set, and cannot modify sensi- tive hardware registers such as %cr3. So the process is constrained to using only its own memory.

## The first system call: exec

Now that we have seen how the kernel provides strong isolation for processes, let’s look at how a user-level process re-enters the kernel to ask for services that it cannot perform itself.

The first action of initcode.S is to invoke the exec system call. 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 unchanged.

Initcode.S (8409) 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 run- ning the program named by $init, which is a pointer to the NUL-terminated string

/init (8422-8424). The other argument is the argv array of command-line arguments;

cpu-

>scheduler+code swtch+code ret+code forkret+code ret+code forkret+code forkret+code main+code

1. >context+code trapret+code swtch+code popal+code popl+code addl+code iret+code initproc+code initcode.S+code allocuvm+code PTE\_U+code userinit+code exec+code SYS\_exec+code T\_SYSCALL+code exec+code

the zero at the end of the array marks its end. If the exec fails and does return, init- code loops calling the exit system call, which definitely should not return (8416-8420).

This code manually crafts the first system call to look like an ordinary system call, which we will see in Chapter 3. 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.

Chapter 2 will cover the implementation of exec in detail, but at a high level it replaces initcode with the /init binary, loaded out of the file system. Now init- code (8400) is done, and the process will run /init instead. Init (8510) 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 re- peats. The system is up.

## Real world

In the real world, one can find both monolithic kernels and microkernels. Many Unix kernels are monolithic. For example, Linux has a monolithic kernel, although some OS functions run as user-level servers (e.g., the windowing system). Kernels such as L4, Minix, QNX are organized as a microkernel with servers, and have seen wide deployment in embedded settings.

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.

## 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. 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?

exit+code

/init+code initcode+code

/init+code

# Chapter 2 Page tables

Page tables are the mechanism through which the operating system controls what

memory addresses mean. They allow xv6 to multiplex the address spaces of different processes onto a single physical memory, and to protect the memories of different pro- cesses. The level of indirection provided by page tables allows many neat tricks. xv6 uses page tables primarily to multiplex address spaces and to protect memory. It also uses a few simple page-table tricks: mapping the same memory (the kernel) in several address spaces, mapping the same memory more than once in one address space (each user page is also mapped into the kernel’s physical view of memory), and guarding a user stack with an unmapped page. The rest of this chapter explains the page tables that the x86 hardware provides and how xv6 uses them. Compared to a real-world operating system, xv6’s design is restrictive, but it does illustrate the key ideas.

## Paging hardware

As a reminder, x86 instructions (both user and kernel) manipulate virtual addresses. The machine’s RAM, or physical memory, is indexed with physical addresses. The x86 page table hardware connects these two kinds of addresses, by mapping each virtual address to a physical address.

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 the address’s top 20 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 2-1](#_bookmark8), 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 ta- ble 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.

Each PTE contains flag bits that tell the paging hardware how the associated vir- tual address is allowed to be used. PTE\_P indicates whether the PTE is present: if it is

page table entries (PTEs)

page

page directory page table pages PTE\_P+code

Virtual address Physical Address

10 10 12 20 12



Dir Table Offset

PPN Offset

20

12

1023

PPN Flags

20

12

1023

1

0

PPN

Flags

Page Table

1

CR3

0

Page Directory

31

Physical Page Number

Page table and page directory entries are identical except for the D bit.

**Figure 2-1**. x86 page table hardware.

121110 9 8 7 6 5 4 3 2 1 0

U

P

W

W T

C D

D A

A V L

P - Present

W - Writable

U - User

WT - 1=Write-through, 0=Write-back CD - Cache Disabled

A - Accessed

D - Dirty (0 in page directory) AVL - Available for system use

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 2-1](#_bookmark8) shows how it all works. The flags and all other page hardware related structures are defined in mmu.h (0700).

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. Instructions use only virtual addresses, which the paging hardware translates to physical addresses, and then sends 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.

## Process address space

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 (1840), because kernel has a more elaborate plan for describing pro- cess address spaces.

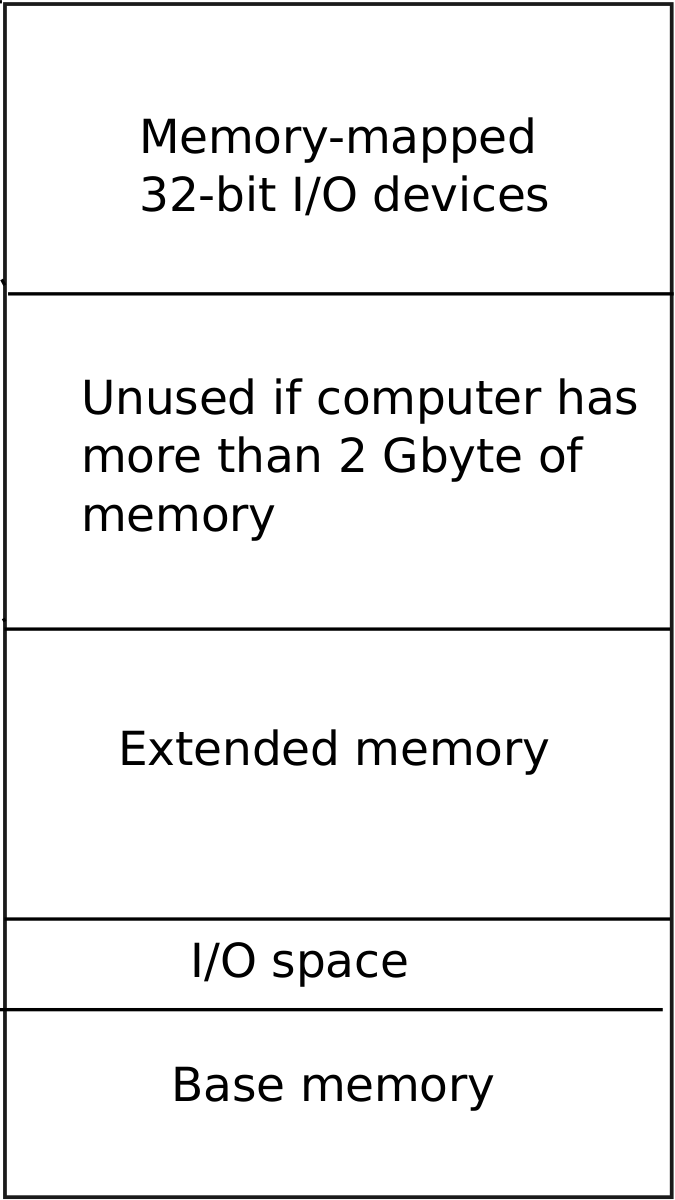
Each process has a separate page table, and xv6 tells the page table hardware to switch page tables when xv6 switches between processes. As shown in [Figure 2-2](#_bookmark9), a process’s user memory starts at virtual address zero and can grow up to KERNBASE, al- lowing a process to address up to 2 gigabytes of memory. The file memlayout.h (0200)

PTE\_W+code PTE\_U+code kvmalloc+code

4 Gbyte

Virtual

Device memory



RW-

0xFE000000

Unused if PHYSTOP is less than 2 Gbyte

Free memory

end

Kernel data

Kernel text

RW--

RW- R--

+ 0x100000

KERNBASE

RW-

4 Gbyte

Physical

0xFE000000

Program data & heap

Unused if computer has more than 2 Gbyte of memory

Extended memory

I/O space Base memory

PHYSTOP

PAGESIZE

User stack

RWU

User data User text

0

RWU RWU

0x100000

640Kbyte

0

**Figure 2-2**. Layout of the virtual address space of a process and the layout of the physical address space. Note that if a machine has more than 2 Gbyte of physical memory, xv6 can use only the memory that fits between KERNBASE and 0xFE00000.

declares the constants for xv6’s memory layout, and macros to convert virtual to physi- cal addresses.

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 ta- ble; these mappings all appear above KERNBASE. It maps virtual addresses KERN- BASE: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 some- times 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 gigabytes of physical memory, because the kernel part of the ad- dress space is 2 gigabytes. Thus, xv6 requires that PHYSTOP be smaller than 2 giga- bytes, even if the computer has more than 2 gigabytes 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. Thus, PHYSTOP must be smaller than two gigabytes - 16 megabytes (for the device memory).

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 use only its own memory. And, each process sees its memory as having contiguous virtual addresses starting at zero, while the process’s physical memory can be non-contiguous. xv6 implements the first by set- ting 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 succes- sive virtual addresses to whatever physical pages happen to be allocated to the process.

## Code: creating an address space

main calls kvmalloc (1840) 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 setup- kvm (1818). 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 (1809) array. The translations include the kernel’s instructions and data, physical memory up to PHYSTOP, and memory ranges which are actually I/O devices. setup- kvm does not install any mappings for the user memory; this will happen later.

mappages (1760) 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, map- pages calls walkpgdir to find the address of the PTE for that address. It then initial- izes 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 (1772).

walkpgdir (1735) mimics the actions of the x86 paging hardware as it looks up the PTE for a virtual address (see [Figure 2-1](#_bookmark8)). walkpgdir uses the upper 10 bits of the virtual address to find the page directory entry (1740). 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 direc- tory. Finally it uses the next 10 bits of the virtual address to find the address of the PTE in the page table page (1753).

## Physical memory allocation

The kernel must 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

PTE\_U+code PTE\_U+code main+code kvmalloc+code setupkvm+code mappages+code kmap+code PHYSTOP+code mappages+code walkpgdir+code walkpgdir+code PHYSTOP+code

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 avail- able for allocation. Each free page’s list element is a struct run (3115). 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 (3119-3123). 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 4 will examine locking in detail.

The function main calls kinit1 and kinit2 to initialize the allocator (3131). The reason for having two calls is that for much of main one cannot use locks or memory above 4 megabytes. The call to kinit1 sets up for lock-less allocation in the first 4 megabytes, and the call to kinit2 enables locking and arranges for more memory to be allocatable. main 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 224 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. kinit1 and kinit2 call freerange to add memory to the free list via per-page calls to kfree. A PTE can only refer to a physical address that is aligned on a 4096-byte boundary (is a multiple of 4096), so freerange uses PGROUNDUP to ensure that it frees only aligned physical ad- dresses. 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 (3164) 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

struct run+code main+code kinit1+code kinit2+code PHYSTOP+code freerange+code kfree+code PGROUNDUP+code

type cast

KERNBASE

argument 0

...

argument N

0

address of argument 0

...

address of argument N address of address of argument 0

argc

0xFFFFFFF

(empty)

|  |
| --- |
| heap |
| stack |
| guard page |
| data |
| text |

PAGESIZE

0

**Figure 2-3**. Memory layout of a user process with its initial stack.

nul-terminated string argv[argc]

argv[0]

argv argument of main

argc argument of main return PC for main

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.

## User part of an address space

[Figure 2-3](#_bookmark10) shows the layout of the user memory of an executing process in xv6. Each user process starts at address 0. The bottom of the address space contains the text for the user program, its data, and its stack. The heap is above the stack so that the heap can expand when the process calls sbrk. Note that the text, data, and stack sections are layed out contiguously in the process’s address space but xv6 is free to use non-contiguous physical pages for those sections. For example, when xv6 expands a process’s heap, it can use any free physical page for the new virtual page and then pro- gram the page table hardware to map the virtual page to the allocated physical page. This flexibility is a major advantage of using paging hardware.

The stack is a single page, and is shown with the initial contents as created by ex- ec. 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. To guard a stack growing off the stack page, xv6 places a guard page right below the stack. The guard page is not mapped and so if the stack runs off the stack page, the hardware will generate an exception because it cannot translate the faulting address. A real- world operating system might allocate more space for the stack so that it can grow be- yond one page.

kalloc+code sbrk+code

## Code: sbrk

Sbrk is the system call for a process to shrink or grow its memory. The system call is implemented by the function growproc (2558). If n is postive, growproc allocates one or more physical pages and maps them at the top of the process’s address space. If n is negative, growproc unmaps one or more pages from the process’s address space and frees the corresponding physical pages. To make these changes, xv6 modifies the pro- cess’s page table. The process’s page table is stored in memory, and so the kernel can update the table with ordinary assignment statements, which is what allocuvm and deallocuvm do. The x86 hardware caches page table entries in a *Translation Look- aside Buffer (TLB)*, and when xv6 changes the page tables, it must invalidate the cached entries. If it didn’t invalidate the cached entries, then at some point later the TLB might use an old mapping, pointing to a physical page that in the mean time has been allocated to another process, and as a result, a process might be able to scribble on some other process’s memory. Xv6 invalidates stale cached entries, by reloading cr3, the register that holds the address of the current page table.

## Code: exec

Exec is the system call that creates the user part of an address space. It initializes the user part of an address space from a file stored in the file system. Exec (6610) opens the named binary path using namei (6623), which is explained in Chapter 6. Then, it 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 (0905), fol- lowed by a sequence of program section headers, struct proghdr (0924). Each progh- dr describes a section of the application that must be loaded into memory; xv6 pro- grams 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 (0902). If the ELF header has the right magic number, exec assumes that the binary is well-formed.

Exec allocates a new page table with no user mappings with setupkvm (6637), allo- cates memory for each ELF segment with allocuvm (6651), and loads each segment into memory with loaduvm (6655). allocuvm checks that the virtual addresses requested is below KERNBASE. loaduvm (1903) uses walkpgdir to find the physical address of the al- located memory at which to write each page of the ELF segment, and readi to read from the file.

The program section header for /init, the first user program created with exec, looks like this:

# objdump -p \_init

\_init: file format elf32-i386 Program Header:

LOAD off 0x00000054 vaddr 0x00000000 paddr 0x00000000 align 2\*\*2

Translation Look- aside Buffer (TLB)

namei+code ELF format

struct elfhdr+code ELF\_MAGIC+code setupkvm+code allocuvm+code loaduvm+code loaduvm+code walkpgdir+code readi+code

/init+code

filesz 0x000008c0 memsz 0x000008cc flags rwx

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. 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.

Exec 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 (2118) function that exec uses to copy arguments to the stack will notice that the destination page is not accessible, and will return –1.

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 re- turns –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 er- ror cases in exec happen during the creation of the image. Once the image is com- plete, exec can install the new image (6701) and free the old one (6702). Finally, exec returns 0.

Exec loads bytes from the ELF file into memory at addresses specified by the ELF file. Users or processes can place whatever addresses they want into an ELF file. Thus exec is risky, because the addresses in the ELF file may refer to the kernel, accidentally or on purpose. The consequences for an unwary kernel could range from a crash to a malicious subversion of the kernel’s isolation mechanisms (i.e., a security exploit). xv6 performs a number of checks to avoid these risks. To understand the importance of these checks, consider what could happen if xv6 didn’t check if(ph.vaddr + ph.memsz < ph.vaddr). This is a check for whether the sum overflows a 32-bit inte- ger. The danger is that a user could construct an ELF binary with a ph.vaddr that points into the kernel, and ph.memsz large enough that the sum overflows to 0x1000. Since the sum is small, it would pass the check if(newsz >= KERNBASE) in allocuvm. The subsequent call to loaduvm passes ph.vaddr by itself, without adding ph.memsz and without checking ph.vaddr against KERNBASE, and would thus copy data from the ELF binary into the kernel. This could be exploited by a user program to run arbi- trary user code with kernel privileges. As this example illustrates, argument checking must be done with great care. It is easy for a kernel developer to omit a crucial check, and real-world kernels have a long history of missing checks whose absence can be ex- ploited by user programs to obtain kernel privileges. It is likely that xv6 doesn’t do a complete job of validating user-level data supplied to the kernel, which a malicious user program might be able to exploit to circumvent xv6’s isolation.

allocuvm+code exec+code ustack+code argv+code argc+code copyout+code

## Real world

Like most operating systems, xv6 uses the paging hardware for memory protec- tion and mapping. Most operating systems use x86’s 64-bit paging hardware (which has 3 levels of translation). 64-bit address spaces allow for a less restrictive memory layout than xv6’s; for example, it would be easy to remove xv6’s limit of 2 gigabytes for physical memory. 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, lazily-allocated pages, and automatically extending stacks. The x86 supports address translation using segmentation (see Appendix B), but xv6 uses segments 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). Implementa- tions 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 gener- al registers that the extra effort required to use segmentation is worthwhile.

Xv6 maps the kernel in the address space of each user process but sets it up so that the kernel part of the address space is inaccessible when the processor is in user mode. This setup is convenient because after a process switches from user space to kernel space, the kernel can easily access user memory by reading memory locations directly. It is probably better for security, however, to have a separate page table for the kernel and switch to that page table when entering the kernel from user mode, so that the kernel and user processes are more separated from each other. This design, for example, would help mitigating side-channels that are exposed by the Meltdown vulnerability and that allow a user process to read arbitrary kernel memory.

On machines with lots of memory it might make sense to use the x86’s 4- megabytes ‘‘super pages.’’ Small pages make sense when physical memory is small, to allow allocation and page-out to disk with fine granularity. For example, if a program uses only 8 kilobytes of memory, giving it a 4 megabytes physical page is wasteful. Larger pages make sense on machines with lots of RAM, and may reduce overhead for page-table manipulation. Xv6 uses super pages in one place: the initial page table (1306). The array initialization sets two of the 1024 PDEs, at indices zero and 512 (KERNBASE>>PDXSHIFT), leaving the other PDEs zero. Xv6 sets the PTE\_PS bit in these two PDEs to mark them as super pages. The kernel also tells the paging hardware to allow super pages by setting the CR\_PSE bit (Page Size Extension) in %cr4.

Xv6 should determine the actual RAM configuration, instead of assuming 224 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 val- ues 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 effi- cient use of limited memory and preparing for unknown future requests; see Knuth. Today people care more about speed than space-efficiency. In addition, a more elabo- rate 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 alloca-

seginit+code CR\_PSE+code

tions as well as large ones.

## Exercises

1. Look at real operating systems to see how they size memory.
2. If xv6 had not used super pages, what would be the right declaration for en-

trypgdir?

1. Write a user program that grows its address space with 1 byte by calling sbrk(1). Run the program and investigate the page table for the program before the call to sbrk and after the call to sbrk. How much space has the kernel allocated? What does the pte for the new memory contain?
2. Modify xv6 so that the pages for the kernel are shared among processes, which reduces memory consumption.
3. Modify xv6 so that when a user program dereferences a null pointer, it will re- ceive a fault. That is, modify xv6 so that virtual address 0 isn’t mapped for user pro- grams.
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. Delete the check if(ph.vaddr + ph.memsz < ph.vaddr) in exec.c, and con- struct a user program that exploits that the check is missing.
6. Change xv6 so that user processes run with only a minimal part of the kernel mapped and so that the kernel runs with its own page table that doesn’t include the user process.
7. How would you improve xv6’s memory layout if xv6 where running on a 64-bit processor?

# Chapter 3

**Traps, interrupts, and drivers**

When running a process, a CPU executes the normal processor loop: read an in- struction, advance the program counter, execute the instruction, repeat. But there are events on which control from a user program must transfer 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 page table entry), 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. Addressing these 3 challenges re- quires 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

There are three cases when control must be transferred from a user program to the kernel. First, a system call: when a user program asks for an operating system service, as we saw at the end of the last chapter. Second, an *exception*: when a program per- forms an illegal action. Examples of illegal actions include divide by zero, attempt to access memory for a page-table entry that is not present, and so on. Third, an *inter- rupt*: when a device generates a signal to indicate that it needs attention from the op- erating 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 in- volved, 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

exception interrupt

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 ex- ceptions 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 inter- rupt 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 exception, xv6 uses the term *trap*, largely because it was the term used by the PDP11/40 and therefore is the conventional Unix term. Furthermore, 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 con- trol 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 de- scriptor.
* Save %esp and %ss in CPU-internal registers, but only if the target segment selec- tor’s PL < CPL.
* Load %ss and %esp from a task segment descriptor.
* Push %ss.

int+code interrupt handler trap

kernel mode user mode

sp from task segment

esp



only present on privilege change

|  |
| --- |
| ss |
| esp |
| eflags |
| cs |
| eip |
| error code |
| (empty) |



**Figure 3-1**. Kernel stack after an int instruction.

* Push %esp.
* Push %eflags.
* Push %cs.
* Push %eip.
* Clear the IF bit in %eflags, but only on an interrupt.
* 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. For example, the check CPL <= DPL allows the kernel to forbid int calls to inappropriate IDT entries such as device interrupt routines. For a user program to execute int, the IDT entry’s DPL must be 3. If the user program doesn’t have the appropriate privilege, then int will result in int 13, which is a gener- al protection fault. As another example, the int instruction cannot use the user stack to save values, because the process may not have a valid stack pointer; instead, the hardware uses the stack specified in the task segment, which is set by the kernel.

[Figure 3-1](#_bookmark12) 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 ta- ble, 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 imple- ment these handlers, and below we will see what xv6 does.

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

## Code: The first system call

Chapter 1 ended with initcode.S invoking a system call. Let’s look at that again

int+code iret+code int+code initcode.S+code

(8414). 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 (3672). We need to arrange that the int in- struction switches the processor from user mode to kernel mode, that the kernel in- vokes the right kernel function (i.e., sys\_exec), and that the kernel can retrieve the arguments for sys\_exec. The next few subsections describe 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 er- rors or attempts to access invalid memory addresses. Xv6 maps the 32 hardware inter- rupts to the range 32-63 and uses interrupt 64 as the system call interrupt.

Tvinit (3367), 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 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 IF 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 (1860) 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 (e.g., a page fault), the processor also pushes an error word. The processor then loads %eip and %cs from the relevant IDT entry.

xv6 uses a Perl script (3250) 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 num- ber, and then jumps to alltraps.

Alltraps (3304) continues to save processor registers: it pushes %ds, %es, %fs,

exec+code sys\_exec+code int+code tvinit main+code idt+code

vectors[i]+code T\_SYSCALL+code IF+code DPL\_USER+code switchuvm+code alltraps+code

cpu->ts.esp0

only present on privilege change

|  |
| --- |
| ss |
| esp |
| eflags |
| cs |
| eip |
|  |
| trapno |
| ds |
| es |
| fs |
| gs |
| eax |
| ecx |
| edx |
| ebx |
| oesp |
| ebp |
| esi |
| edi |
| (empty) |

esp



p->kstack



**Figure 3-2**. The trapframe on the kernel stack

%gs, and the general-purpose registers (3305-3310). The result of this effort is that the kernel stack now contains a struct trapframe (0602) containing the processor regis- ters at the time of the trap (see [Figure 3-2](#_bookmark13)). 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 2, that userinit built a trapframe by hand to achieve this goal (see [Figure 1-4](#_bookmark6)).

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 set- ting 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 (3313-3315).

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 (3318). Then it calls trap (3319). After trap returns, alltraps pops the argument off the stack by adding to the stack pointer (3320) and then starts execut-

alltraps+code alltraps+code alltraps+code trap+code alltraps+code

ing the code at label trapret. We traced through this code in Chapter 2 when the first user process ran it to exit to user space. The same sequence happens here: pop- ping 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 (3401) 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 proc->killed checks in Chapter 5.

After checking for a system call, trap looks for hardware interrupts (which we dis- cuss 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 proc->killed to remember to clean up the user process. We will look at how xv6 does this cleanup in Chapter 5.

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 (3701). 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 (3507), and syscall will invoke the SYS\_exec’th entry of the system call table, which corre- sponds 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 regis- ters. Thus, when exec returns, it will return the value that the system call handler re- turned (3708). 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, argptr, argstr,

trapret+code iret+code trap+code

tf->trapno+code T\_SYSCALL+code syscall+code

proc->killed+code trap+code panic+code trap+code syscall+code SYS\_exec+code

cp->tf+code syscall+code

and argfd retrieve the *n*’th system call argument, as either an integer, pointer, a string, or a file descriptor. argint uses the user-space %esp register to locate the *n’th* argu- ment: %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 lies within 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. The kernel, however, can derefence any address that the user might have passed, so it must check explicitly that the address is below p->sz.

argptr fetches the *n*th system call argument and checks that this argument is a

valid user-space pointer. Note that two checks occur during a call to argptr. First, the user stack pointer is checked during the fetching of the argument. Then the argument, itself a user pointer, is checked.

argstr 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.

Finally, argfd (6071) uses argint to retrieve a file descriptor number, checks if it is valid file descriptor, and returns the corresponding struct file.

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 2, sys\_exec uses these functions to get at its ar- guments.

## Code: Interrupts

Devices on the motherboard can generate interrupts, and xv6 must set up the hardware to handle these interrupts. Devices usually interrupt in order to tell the ker- nel that some hardware event has occured, such as I/O completion. Interrupts are usually optional in the sense that the kernel could instead periodically check (or "poll") the device hardware to check for new events. Interrupts are preferable to polling if the events are relatively rare, so that polling would waste CPU time. Interrupt handling shares some of the code already needed for system 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 atten- tion (e.g., the user has typed a character on the keyboard). We must program the de- vice 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 hard- ware 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 process- es. The choice of 100 times per second allows for decent interactive performance while not swamping the processor with handling interrupts.

argint+code fetchint+code p->sz+code argptr+code argstr+code argfd+code

Like the x86 processor itself, PC motherboards have evolved, and the way inter- rupts are provided has evolved too. The early boards had a simple programmable in- terrupt controler (called the PIC). 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 sent to it, and there must be a method for routing in- terrupts 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: it ignores in- terrupts from the PIC, and configures the IOAPIC and local APIC.

The IO APIC has a table and the processor can program entries in the table through memory-mapped I/O. 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 (8274). 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 in- terrupts independently. Xv6 sets it up in lapicinit (7408). The key line is the one that programs the timer (7421). This line tells the LAPIC to periodically generate an inter- rupt at IRQ\_TIMER, which is IRQ 0. Line (7451) 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 (9112) and the other processors (1124). The scheduler on each processor enables interrupts (2766). To control that certain code fragments are not interrupted, xv6 disables interrupts during these code fragments (e.g., see switchuvm (1860)).

The timer interrupts through vector 32 (which xv6 chose to handle IRQ 0), which xv6 setup in idtinit (1255). 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 clear 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 for a timer interrupt does just two things: increment the ticks variable (3417), and call wakeup. The latter, as we will see in Chapter 5, may cause the interrupt to return in a different process.

## Drivers

A *driver* is the code in an operating system that manages a particular device: it tells the device hardware to perform operations, configures the device to generate interrupts when done, and handles the resulting interrupts. 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. The disk driver copies data from and

lapicinit+code IRQ\_TIMER,+code IF+code

cli+code sti+code switchuvm+code idtinit+code trap+code wakeup+code driver

back to the disk. Disk hardware traditionally presents the data on the disk as a num- bered sequence of 512-byte *blocks* (also called *sectors*): sector 0 is the first 512 bytes, sector 1 is the next, and so on. The block size that an operating system uses for its file system maybe different than the sector size that a disk uses, but typically the block size is a multiple of the sector size. Xv6’s block size is identical to the disk’s sector size. To represent a block xv6 has a structure struct buf (3850). The data stored in this struc- ture 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 con- troller. 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 de- tails of a particular piece of hardware.

Xv6 represent file system blocks using struct buf (3850). BSIZE (4055) is identical to the IDE’s sector size and thus 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. Although the xv6 file sys- tem chooses BSIZE to be identical to the IDE’s sector size, the driver can handle a BSIZE that is a multiple of the sector size. Operating systems often use bigger blocks than 512 bytes to obtain higher disk throughput.

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 kernel initializes the disk driver at boot time by calling ideinit (4251) from main (1232). Ideinit calls ioapicenable to enable the IDE\_IRQ interrupt (4256). The call to ioapicenable enables the interrupt 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 (4257) to wait for the disk to be able to accept commands. A PC motherboard presents the sta- tus bits of the disk hardware on I/O port 0x1f7. Idewait (4238) 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 (4259-4266). 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.

block sector

struct buf+code B\_VALID+code B\_DIRTY+code ideinit+code main+code ioapicenable+code IDE\_IRQ+code ideinit+code idewait+code IDE\_BSY+code IDE\_DRDY+code iderw+code B\_DIRTY+code B\_VALID+code iderw+code

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 (see Appendix B). 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 fin- ished. 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 (4354) adds the buffer b to the end of the queue (4367-4371). If the buffer is at the front of the queue, iderw must send it to the disk hardware by calling idestart (4326-4328); otherwise the buffer will be started once the buffers ahead of it are taken care of.

Idestart (4274) 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 (4296). idestart moves the data to a buffer in the disk controller using the outsl in- struction; using CPU instructions to move data to/from device hardware is called pro- grammed I/O. Eventually the disk hardware will raise an interrupt to 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 idestart has detailed knowl- edge 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 challeng- ing.

polling

busy waiting iderw+code idestart+code idestart+code iderw+code trap+code ideintr+code insl+code B\_VALID+code B\_DIRTY+code

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 yields the CPU for other processes by sleeping, waiting for the interrupt handler to record in the buffer’s flags that the operation is done (4378-4379). 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 (3424). Ideintr (4304) 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 from a buffer in the disk controller into memory with insl (4317-4319). Now the buffer is ready: ideintr sets B\_VALID, clears B\_DIRTY, and wakes up any process sleeping on the buffer (4321-4324). Finally, ideintr must pass the next waiting buffer to the disk (4326-4328).

## Real world

Supporting all the devices on a PC motherboard in its full glory is much work, be- cause there are many devices, the devices have many features, and the protocol be- tween device and driver can be complex. In many operating systems, the drivers to- gether 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 hard- ware uses interrupts to notify the operating system of status changes. Modern disk controllers typically accept a *batch* of disk requests at a time and even reorder them to make most efficient use of the disk arm. When disks were simpler, operating systems 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 disk works very differently from a traditional mechanical disk, both devices provide block-based interfaces and read- ing/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 pack- ets, 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 programmed I/O (insl, outsl). DMA allows the device direct access to physical memory. The driver gives the device the physical address of the buffer’s data and the device copies directly to or from main memory, interrupting once the copy is complete. DMA is faster and more efficient than programmed I/O and is less taxing for the CPU’s memory caches.

Some drivers dynamically switch between polling and interrupts, because using interrupts can be expensive, but using polling can introduce delay until the driver pro- cesses an event. For example, 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 imme- diately when a new packet arrives.

The IDE driver routes interrupts statically to a particular processor. Some drivers configure the IO APIC to route interrupts to multiple processors to spread out the work of processing packets. 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 an- other 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 a program 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 program then sends the data over the network, the data is copied twice more: from user space to kernel space and from kernel space to the network device. To support applications for which efficiency is im- portant (e.g., serving popular images on the Web), operating systems use special code paths to avoid copies. As one example, in real-world operating systems, buffers typical- ly match the hardware page size, so that read-only copies can be mapped into a pro- cess’s address space using the paging hardware, without any copying.

## Exercises

batch

* 1. Set a breakpoint at the first instruction of syscall to catch the very first sys- tem call (e.g., br syscall). What values are on the stack at this point? Explain the out- put 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.).

* 1. Add a new system call to get the current UTC time and return it to the user program. You may want to use the helper function, cmostime (7552), to read the real time clock. The file date.h contains the definition of the struct rtcdate (0950), which you will provide as an argument to cmostime as a pointer.
  2. Write a driver for a disk that supports the SATA standard (search for SATA on the Web). Unlike IDE, SATA isn’t obsolete. Use SATA’s tagged command queuing to issue many commands to the disk so that the disk internally can reorder commands to obtain high performance.
  3. Add simple driver for an Ethernet card.

# Chapter 4 Locking

Xv6 runs on multiprocessors: computers with multiple CPUs executing indepen-

dently. These multiple CPUs share physical RAM, and xv6 exploits the sharing to maintain data structures that all CPUs read and write. This sharing raises the possibil- ity of one CPU reading a data structure while another CPU is mid-way through up- dating it, or even multiple CPUs updating the same data simultaneously; without care- ful design such parallel access is likely to yield incorrect results or a broken data struc- ture. Even on a uniprocessor, an interrupt routine that uses the same data as some in- terruptible code could damage the data if the interrupt occurs at just the wrong time.

Any code that accesses shared data concurrently must have a strategy for main- taining correctness despite concurrency. The concurrency may arise from accesses by multiple cores, or by multiple threads, or by interrupt code. xv6 uses a handful of sim- ple concurrency control strategies; much more sophistication is possible. This chapter focuses on one of the strategies used extensively in xv6 and many other systems: the *lock*.

lock

A lock provides mutual exclusion, ensuring that only one CPU at a time can hold the lock. If a lock is associated with each shared data item, and the code always holds the associated lock when using a given item, then we can be sure that the item is used from only one CPU at a time. In this situation, we say that the lock protects the data item.

The rest of this chapter explains why xv6 needs locks, how xv6 implements them, and how it uses them. A key observation will be that if you look at some code in xv6, you must ask yourself if another processor (or interrupt) could change the intended behavior of the code by modifying data (or hardware resources) it depends on. You must keep in mind that a single C statement can be several machine instructions and thus another processor or an interrupt may muck around in the middle of a C state- ment. You cannot assume that lines of code on the page are executed atomically. Concurrency makes reasoning about correctness much more difficult.

## Race conditions

As an example of 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 out- standing disk requests (4226) and processors may add new requests to the list concur- rently (4354). If there were no concurrent requests, you might implement the linked list as follows:

CPU 1

15 16

Memory



l->next list

l->next list

CPU2

15 16



**Figure 4-1**. Example race

Time

* + 1. struct list {
    2. int data;
    3. struct list \*next; 4 };

5

6 struct list \*list = 0; 7

1. void
2. insert(int data)

10 {

11 struct list \*l; 12

1. l = malloc(sizeof \*l);
2. l->data = data;
3. l->next = list;
4. list = l; 17 }

This implementation is correct if executed in isolation. However, the code is not cor- rect if more than one copy executes concurrently. If two CPUs execute insert at the same time, it could happen that both execute line 15 before either executes 16 (see [Figure 4-1](#_bookmark15)). If this happens, there will now be two list nodes with next set to the for- mer 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.

The lost update at line 16 is an example of a *race condition*. A race condition is a

situation in which a memory location is accessed concurrently, and at least one access is a write. A race is often a sign of a bug, either a lost update (if the accesses are writes) or a read of an incompletely-updated data structure. The outcome of a race depends on the exact timing of the two CPUs involved and how their memory opera- tions are ordered by the memory system, which can make race-induced errors difficult to reproduce and debug. For example, adding print statements while debugging in-

race condition

sert might change the timing of the execution enough to make the race disappear.

The usual 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 im- possible. The correctly locked version of the above code adds just a few lines (not numbered):

6 struct list \*list = 0; struct lock listlock;

mutual exclusion critical section serializing

7

1. void
2. insert(int data)

10 {

1. struct list \*l;
2. l = malloc(sizeof \*l);
3. l->data = data;

14

acquire(&listlock);

1. l->next = list;
2. list = l; release(&listlock);

17 }

The sequence of instructions between acquire and release is often called a *critical section*, and the lock protects list.

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 struc- tures 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 ex- ample, 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 violates this invariant temporarily: in line 15, l points to the next list element, but list does not point at l yet (reestablished at line 16). The race condition we ex- amined 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 in the critical section, so that no CPU will execute a data structure operation when the data structure’s invari- ants do not hold.

You can think of locks as *serializing* concurrent critical sections so that they run

one at a time, and thus preserve invariants (assuming they are correct in isolation). You can also think of critical sections as being atomic with respect to each other, so that a critical section that obtains the lock later sees only the complete set of changes from earlier critical sections, and never sees partially-completed updates.

Note that it would also be correct to move up acquire to earlier in insert. For example, it is fine to move the call to acquire up to before line 12. This may reduce paralellism because then the calls to malloc are also serialized. The section "Using locks" below provides some guidelines for where to insert acquire and release invo- cations.

## Code: Locks

Xv6 has two types of locks: spin-locks and sleep-locks. We’ll start with spin-locks. Xv6 represents a spin-lock as a struct spinlock (1501). The important 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

1. void
2. acquire(struct spinlock \*lk) 23 {

24 for(;;) {

1. if(!lk->locked) {
2. lk->locked = 1;
3. break;

28 }

29 }

30 }

Unfortunately, this implementation does not guarantee mutual exclusion on a multi- processor. It could happen that two CPUs simultaneously reach line 25, see that lk-

>locked is zero, and then both grab the lock by executing line 26. 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 x86 instruction, xchg (0569). In one atomic operation, xchg swaps a word in memory with the contents of a register. The function acquire (1574) repeats this xchg instruction in a loop; each iter- ation atomically reads lk->locked and sets it to 1 (1581). If the lock is already 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. If a process forgets to re- lease a lock, 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 (1602) is the opposite of acquire: it clears the debugging fields and then releases the lock. The function uses an assembly instruction to clear locked, because clearing this field should be atomic so that the xchg instruction won’t see a subset of the 4 bytes that hold locked updated. The x86 guarantees that a 32-bit movl updates all 4 bytes atomically. Xv6 cannot use a regular C assignment, because the C language specification does not specify that a single assignment is atomic.

Xv6’s implementation of spin-locks is x86-specific, and xv6 is thus not directly portable to other processors. To allow for portable implementations of spin-locks, the C language supports a library of atomic instructions; a portable operating system would use those instructions.

## Code: Using locks

struct

spinlock+code acquire+code atomic xchg+code acquire+code release+code

Xv6 uses locks in many places to avoid race conditions. A simple example is in the IDE driver (4200). As mentioned in the beginning of the chapter, iderw (4354) has a queue of disk requests and processors may add new requests to the list concurrently (4369). To protect this list and other invariants in the driver, iderw acquires the ide- lock (4365) and releases it at the end of the function.

Exercise 1 explores how to trigger the IDE driver 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 un- likely 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 memory locations, typically all of them 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 parallelism isn’t important, then one could arrange to have only a single thread and not worry about locks. A simple kernel can do this on a multipro- cessor by having a single lock that must be acquired on entering the kernel and re- leased on exiting the kernel (though system calls such as pipe reads or wait would pose a problem). Many uniprocessor operating systems have been converted to run on multiprocessors using this approach, sometimes called a ‘‘giant kernel lock,’’ but the ap- proach sacrifices parallelism: 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 simultane- ously.

Ultimately, the choice of lock granularity is an exercise in parallel programming. Xv6 uses a few coarse data-structure specific locks (see [Figure 4-2](#_bookmark16)). For example, xv6 has a lock that protects the whole process table and its invariants, which are described in Chapter 5. 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 pro- ceed in parallel. However, it complicates operations that have invariants over the whole process table, since they might have to acquire several locks. Subsequent chap- ters will discuss how each part of xv6 deals with concurrency, illustrating how to use locks.

## Deadlock and lock ordering

If a code path through the kernel must hold several locks at the same time, it is im- portant 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 need locks A and B, but code path 1 acquires locks in the order A then B, and the other path acquires them in the order B

iderw+code idelock+code

**Lock Description**

bcache.lock Protects allocation of block buffer cache entries

cons.lock Serializes access to console hardware, avoids intermixed output ftable.lock Serializes allocation of a struct file in file table

icache.lock Protects allocation of inode cache entries

idelock Serializes access to disk hardware and disk queue kmem.lock Serializes allocation of memory

log.lock Serializes operations on the transaction log pipe’s p->lock Serializes operations on each pipe

ptable.lock Serializes context switching, and operations on proc->state and proctable tickslock Serializes operations on the ticks counter

inode’s ip->lock Serializes operations on each inode and its content buf’s b->lock Serializes operations on each block buffer

**Figure 4-2**. Locks in xv6

then A. This situation can result in a deadlock if two threads execute the code paths concurrently. Suppose thread T1 executes code path 1 and acquires lock A, and thread T2 executes code path 2 and acquires lock B. Next T1 will try to acquire lock B, and T2 will try to acquire lock A. Both acquires will block indefinitely, because in both cases the other thread holds the needed lock, and won’t release it until its acquire re- turns. To avoid such deadlocks, all code paths must acquire locks in the same order. The need for a global lock acquisition order means that locks are effectively part of each function’s specification: callers must invoke functions in a way that causes locks to be acquired in the agreed-on order.

Xv6 has many lock-order chains of length two involving the ptable.lock, due to the way that sleep works as discussed in Chapter 5. For example, ideintr holds the ide lock while calling wakeup, which acquires the ptable lock. The file system code contains xv6’s longest lock chains. For example, creating a file requires simultaneously holding a lock on the directory, a lock on the new file’s inode, a lock on a disk block buffer, idelock, and ptable.lock. To avoid deadlock, file system code always acquires locks in the order mentioned in the previous sentence.

## Interrupt handlers

Xv6 uses spin-locks in many situations to protect data that is used by both interrupt handlers and threads. For example, a timer interrupt might (3414) increment ticks at about the same time that a kernel thread reads ticks in sys\_sleep (3823). The lock tickslock serializes the two accesses.

Interrupts can cause concurrency even on a single processor: if interrupts are en- abled, 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 situ- ation, 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.

ideintr+code wakeup+code ptable+code ticks+code sys\_sleep+code tickslock+code iderw+code idelock+code ideintr+code

To avoid this situation, if a spin-lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled. Xv6 is more conservative: when a processor enters a spin-lock critical section, xv6 always ensures interrupts are disabled on that processor. Interrupts may still occur on other processors, so an interrupt’s ac- quire can wait for a thread to release a spin-lock; just not on the same processor.

xv6 re-enables interrupts when a processor holds no spin-locks; it must do a little book-keeping to cope with nested critical sections. acquire calls pushcli (1667) and release calls popcli (1679) to track the nesting level of locks on the current processor. When that count reaches zero, popcli restores the interrupt enable state that existed at the start of the outermost critical section. The cli and sti functions execute the x86 interrupt disable and enable instructions, respectively.

It is important that acquire call pushcli before the xchg that might acquire the lock (1581). 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 (1581).

## Instruction and memory ordering

This chapter has assumed that code executes in the order in which the code ap- pears in the program. Many compilers and processors, however, execute code 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 instructions A and B are not dependent on each other and start in- struction B before A so that it will be completed when the processor completes A. A compiler may perform a similar re-ordering by emitting instruction B before instruc- tion A in the executable file. Concurrency, however, may expose this reordering to software, which can lead to incorrect behavior.

For example, in this code for insert, it would be a disaster if the compiler or processor caused the effects of line 4 (or 2 or 5) to be visible to other cores after the effects of line 6:

* 1. l = malloc(sizeof \*l);
  2. l->data = data;
  3. acquire(&listlock);
  4. l->next = list;
  5. list = l;
  6. release(&listlock);

If the hardware or compiler would re-order, for example, the effects of line 4 to be vis- ible after line 6, then another processor can acquire listlock and observe that list points to l, but it won’t observe that l->next is set to the remainder of the list and won’t be able to read the rest of the list.

To tell the hardware and compiler not to perform such re-orderings, xv6 uses

sync\_synchronize(), in both acquire and release. \_sync\_synchronize() is a memory barrier: it tells the compiler and CPU to not reorder loads or stores across

pushcli+code popcli+code acquire+code xchg+code release+code popcli+code xchg+code

the barrier. Xv6 worries about ordering only in acquire and release, because con- current access to data structures other than the lock structure is performed between acquire and release.

## Sleep locks

Sometimes xv6 code needs to hold a lock for a long time. For example, the file system (Chapter 6) keeps a file locked while reading and writing its content on the disk, and these disk operations can take tens of milliseconds. Efficiency demands that the processor be yielded while waiting so that other threads can make progress, and this in turn means that xv6 needs locks that work well when held across context switches. Xv6 provides such locks in the form of *sleep-locks*.

Xv6 sleep-locks support yielding the processor during their critical sections. This property poses a design challenge: if thread T1 holds lock L1 and has yielded the pro- cessor, and thread T2 wishes to acquire L1, we have to ensure that T1 can execute while T2 is waiting so that T1 can release L1. T2 can’t use the spin-lock acquire func- tion here: it spins with interrupts turned off, and that would prevent T1 from running. To avoid this deadlock, the sleep-lock acquire routine (called acquiresleep) yields the processor while waiting, and does not disable interrupts.

acquiresleep (4622) uses techniques that will be explained in Chapter 5. At a high level, a sleep-lock has a locked field that is protected by a spinlock, and ac- quiresleep’s call to sleep atomically yields the CPU and releases the spin-lock. The result is that other threads can execute while acquiresleep waits.

Because sleep-locks leave interrupts enabled, they cannot be used in interrupt handlers. Because acquiresleep may yield the processor, sleep-locks cannot be used inside spin-lock critical sections (though spin-locks can be used inside sleep-lock criti- cal sections).

Xv6 uses spin-locks in most situations, since they have low overhead. It uses sleep-locks only in the file system, where it is convenient to be able to hold locks across lengthy disk operations.

## Limitations of locks

Locks often solve concurrency problems cleanly, but there are times when they are awkward. Subsequent chapters will point out such situations in xv6; this section out- lines some of the problems that come up.

Sometimes a function uses data which must be guarded by a lock, but the func- tion is called both from code that already holds the lock and from code that wouldn’t otherwise need the lock. One way to deal with this is to have two variants of the function, one that acquires the lock, and the other that expects the caller to already hold the lock; see wakeup1 for an example (2953). Another approach is for the function to require callers to hold the lock whether the caller needs it or not, as with sched (2758). Kernel developers need to be aware of such requirements.

It might seem that one could simplify situations where both caller and callee need a lock by allowing *recursive locks*, so that if a function holds a lock, any function it

sleep-locks recursive locks

calls is allowed to re-acquire the lock. However, the programmer would then need to reason about all combinations of caller and callee, because it will no longer be the case that the data structure’s invariants always hold after an acquire. Whether recursive locks are better than xv6’s use of conventions about functions that require a lock to be held is not clear. The larger lesson is that (as with global lock ordering to avoid dead- lock) lock requirements sometimes can’t be private, but intrude themselves on the in- terfaces of functions and modules.

A situation in which locks are insufficient is when one thread needs to wait for another thread’s update to a data structure, for example when a pipe’s reader waits for some other thread to write the pipe. The waiting thread cannot hold the lock on the data, since that would prevent the update it is waiting for. Instead, xv6 provides a sepa- rate mechanism that jointly manages the lock and event wait; see the description of sleep and wakeup in Chapter 5.

## Real world

Concurrency primitives and parallel programming are active areas 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.

Most operating systems support POSIX threads (Pthreads), which allow a user process to have several threads running concurrently on different processors. Pthreads has support for user-level locks, barriers, etc. Supporting Pthreads requires support from the operating system. For example, it should be the case that if one pthread blocks in a system call, another pthread of the same process should be able to run on that processor. As another example, if a pthread changes its process’s address space (e.g., grow or shrink it), the kernel must arrange that other processors that run threads of the same process update their hardware page tables to reflect the change in the ad- dress space. On the x86, this involves shooting down the *Translation Look-aside Buffer (TLB)* of other processors using inter-processor interrupts (IPIs).

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

Locks can be expensive if many processors try to acquire the same lock at the same time. If one processor has a lock cached in its local cache, and another proces- sor must acquire the lock, then the atomic instruction to update the cache line that holds the lock must move the line from the one processor’s cache to the other proces- sor’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. For example, it is possible to implement a linked 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. Lock-free programming is more complicated, however, than programming locks; for example, one must worry about in-

Translation Look- aside Buffer (TLB)

struction and memory reordering. Programming with locks is already hard, so xv6 avoids the additional complexity of lock-free programming.

## Exercises

1. 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.
2. Remove the xchg in acquire. Explain what happens when you run xv6?
3. Write a parallel program using POSIX threads, which is supported on most op- erating systems. For example, implement a parallel hash table and measure if the num- ber of puts/gets scales with increasing number of cores.
4. Implement a subset of Pthreads in xv6. That is, implement a user-level thread library so that a user process can have more than 1 thread and arrange that these threads can run in parallel on different processors. Come up with a design that cor- rectly handles a thread making a blocking system call and changing its shared address space.

# Chapter 5 Scheduling

Any operating system is likely to run with more processes than the computer has

processors, so a plan is needed to time-share the processors among the processes. Ide- ally the sharing would be transparent to user processes. A common approach is to provide each process with the illusion that it has its own virtual processor by *multi- plexing* the processes onto the hardware processors. This chapter explains how xv6 achieves this multiplexing.

## Multiplexing

Xv6 multiplexes by switching each processor from one process to another in two situations. First, xv6’s sleep and wakeup mechanism switches when a process waits for device or pipe I/O to complete, or waits for a child to exit, or waits in the sleep sys- tem call. Second, xv6 periodically forces a switch when a process is executing user in- structions. This multiplexing creates the illusion that each process has its own CPU, just as xv6 uses the memory allocator and hardware page tables to create the illusion that each process has its own memory.

Implementing multiplexing poses a few challenges. First, how to switch from one process to another? Although the idea of context switching is simple, the implementa- tion is some of the most opaque code in xv6. Second, how to switch transparently to user processes? Xv6 uses the standard technique of driving context switches with timer interrupts. Third, many CPUs may be switching among processes concurrently, and a locking plan is necessary to avoid races. Fourth, a process’s memory and other resources must be freed when the process exits, but it cannot do all of this itself be- cause (for example) it can’t free its own kernel stack while still using it. Finally, each core of a multi-core machine must remember which process it is executing so that sys- tem calls affect the correct process’s kernel state. 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 exam- ple, a parent process may need to wait for one of its children to exit, or a process reading 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

multiplexing



user space

shell

cat

save

swtch

swtch

restore

kernel space

kstack

shell

kstack kstack scheduler cat

Kernel

**Figure 5-1**. Switching from one user process to another. In this example, xv6 runs with one CPU (and thus one scheduler thread).

[Figure 5-1](#_bookmark18) outlines the steps involved in switching from one user process to an- other: a user-kernel transition (system call or interrupt) to the old process’s kernel thread, a context switch to the current CPU’s scheduler thread, a context switch to a new process’s kernel thread, and a trap return to the user-level process. The xv6 scheduler has its own thread (saved registers and stack) because it is sometimes not safe for it execute on any process’s kernel stack; we’ll see an example in exit. In this section we’ll examine the mechanics of switching between a kernel thread and a sched- uler thread.

Switching from one thread to another involves saving the old thread’s CPU regis- ters, and restoring the 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.

The function swtch performs the saves and restores for a thread switch. swtch doesn’t directly know about threads; it just saves and restores register sets, called *con- texts*. When it is time for a process to give up the CPU, the process’s kernel thread calls 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 registers onto the stack and saves the stack point- er in \*old. Then swtch copies new to %esp, pops previously saved registers, and re- turns.

Let’s follow a user process through swtch into the scheduler. We saw in Chapter 3 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 (2822).

Swtch (3052) starts by copying its arguments from the stack to the caller-saved reg- isters %eax and %edx (3060-3061); 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-saved registers need to be saved; the convention on the x86 is that these are %ebp, %ebx, %esi,

swtch+code contexts

struct context+code trap+code yield+code sched+code swtch+code

cpu-

>scheduler+code swtch+code

%edi, and %esp. Swtch pushes the first four explicitly (3064-3067); it saves the last im- plicitly as the struct context\* written to \*old (3070). There is one more important register: the program counter %eip. It has already been saved on the stack by the call instruction that invoked swtch. 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 (3071). 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 (3074-3078). Because swtch has changed the stack pointer, the values re- stored 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 (2781). When the swtch we have been tracing returns, it returns not to sched but to sched- uler, and its stack pointer points at the current CPU’s scheduler stack.

## Code: Scheduling

The last section looked at the low-level details of swtch; now let’s take swtch as a given and examine switching from a process through the scheduler to another process. A process that wants to give up the CPU must acquire the process table lock pt- able.lock, release any other locks it is holding, update its own state (proc->state), and then call sched. Yield (2828) follows this convention, as do sleep and exit, which we will examine later. Sched double-checks those conditions (2813-2818) 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 re- turns on the scheduler’s stack as though scheduler’s swtch had returned (2781). 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; usually the thread that acquires a lock is also responsible for releasing the lock, which makes it easier to reason about correct- ness. For context switching it is necessary to break this convention because pt- able.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 pt- able.lock were not held during swtch: a different CPU might decide to run the pro- cess 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 some kernel thread that previously called sched. Thus, if one were to print out the line numbers where xv6 switches threads, one would observe the following simple pattern: (2781), (2822), (2781), (2822), and so on. The procedures in which this stylized switching between two

swtch+code sched+code swtch+code cpu-

>scheduler+code swtch+code scheduler+code swtch+code ptable.lock+code sched+code sleep+code exit+code sched+code swtch+code

cpu-

>scheduler+code scheduler+code ptable.lock+code swtch+code ptable.lock+code swtch+code yield+code

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 call to swtch does not end up in sched. We saw this case in Chapter 2: when a new process is first scheduled, it begins at forkret (2853). Forkret exists to release the ptable.lock; otherwise, the new process could start at trapret.

Scheduler (2758) runs a simple loop: find a process to run, run it until it yields, 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 im- portant 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 (2774-2781).

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, a timer inter- rupt’s yield must be able to 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, an idle CPU’s scheduler must be able to 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 pro- cess’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.

coroutines sched+code scheduler+code swtch+code sched+code forkret+code ptable.lock+code scheduler+code ptable.lock+code RUNNABLE+code switchuvm+code swtch+code ptable.lock+code yield+code RUNNABLE+code scheduler+code

p->context+code ptable.lock+code ptable.lock+code exit+code wait+code

## Code: mycpu and myproc

xv6 maintains a struct cpu for each processor, which records the process cur- rently running on the processor (if any), the processor’s unique hardware identifier (apicid), and some other information. The function mycpu (2437) returns the current processor’s struct cpu. mycpu does this by reading the processor identifier from the local APIC hardware and looking through the array of struct cpu for an entry with that identifier. The return value of mycpu is fragile: if the timer were to interrupt and cause the thread to be moved to a different processor, the return value would no longer be correct. To avoid this problem, xv6 requires that callers of mycpu disable in- terrupts, and only enable them after they finish using the returned struct cpu.

The function myproc (2457) returns the struct proc pointer for the process that is running on the current processor. myproc disables interrupts, invokes mycpu, fetches the current process pointer (c->proc) out of the struct cpu, and then enables inter- rupts. If there is no process running, because the the caller is executing in scheduler, myproc returns zero. The return value of myproc is safe to use even if interrupts are enabled: if a timer interrupt moves the calling process to a different processor, its struct proc pointer will stay the same.

## Sleep and wakeup

Scheduling and locks help conceal the existence of one process from another, but so far we have no abstractions that help processes intentionally interact. 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 similar 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 that feeds commands from processes to the IDE driv- er (see Chapter 3), but abstracts away all IDE-specific code. The queue allows one process to send a nonzero pointer to another process. If there were only one sender and one receiver, and they executed on different CPUs, and the compiler didn’t opti- mize too agressively, this implementation would be correct:

1. struct q {
2. void \*ptr;

102 };

103

1. void\*
2. send(struct q \*q, void \*p)

106 {

107 while(q->ptr != 0)

108 ;

109 q->ptr = p;

110 }

111

struct cpu+code mycpu+code myproc+code sequence

coordination conditional

synchronization

1. void\*
2. 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 modify q->ptr, but send only writes the pointer when it is zero and recv only writes the pointer when it is nonzero, so no up- dates are lost.

The implementation above 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 re- ceiver’s CPU could find more productive work than *busy waiting* by repeatedly *polling* q->ptr. Avoiding busy waiting requires a way for the receiver to yield the CPU and resume only when send delivers 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 pro- cesses are waiting on chan, wakeup does nothing. We can change the queue imple- mentation to use sleep and wakeup:

1. void\*
2. send(struct q \*q, void \*p)

203 {

204 while(q->ptr != 0)

205 ;

1. q->ptr = p;
2. wakeup(q); /\* wake recv \*/

208 }

209

1. void\*
2. recv(struct q \*q)

212 {

213 void \*p;

214

1. while((p = q->ptr) == 0)
2. 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 5-2](#_bookmark19)). Suppose that recv finds that q->ptr == 0 on line 215. While recv is between lines 215 and

busy waiting polling sleep+code wakeup+code chan+code wait channel

recv

215

test

216

sleep

wait for wakeup forever

Time



send

206 207

204

205

store p

wakeup

test

spin forever

**Figure 5-2**. Example lost wakeup problem

216, 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 execut- ing 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 wait 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. One incorrect way of protecting the invariant would be to modify the code for recv as follows:

1. struct q {
2. struct spinlock lock;
3. void \*ptr;

303 };

304

1. void\*
2. send(struct q \*q, void \*p)

307 {

1. acquire(&q->lock);
2. while(q->ptr != 0)

310 ;

1. q->ptr = p;
2. wakeup(q);
3. release(&q->lock);

314 }

315

1. void\*
2. recv(struct q \*q)

318 {

319 void \*p;

320

1. acquire(&q->lock);
2. while((p = q->ptr) == 0)
3. sleep(q);

324 q->ptr = 0;

1. release(&q->lock);
2. return p;

327 }

One might hope that this version of recv would avoid the lost wakeup because the

deadlocked

lock prevents send from executing between lines 322 and 323. It does that, but it also deadlocks: recv holds the lock while it sleeps, so the sender will block forever waiting for the lock.

We’ll fix the preceding scheme by changing sleep’s interface: the caller must pass the lock to sleep so it can release the lock after the calling process is marked as asleep and waiting on the sleep channel. The lock will force a concurrent send to wait until the receiver has finished putting itself to sleep, so that the wakeup will find the sleeping receiver and wake it up. Once the receiver is awake again sleep reacquires the lock before returning. Our new correct scheme is useable as follows:

1. struct q {
2. struct spinlock lock;
3. void \*ptr;

403 };

404

405 void\*

406 send(struct q \*q, void \*p)

407 {

408 acquire(&q->lock);

409 while(q->ptr != 0)

410 ;

411 q->ptr = p;

412 wakeup(q);

413 release(&q->lock);

414 }

415

416 void\*

417 recv(struct q \*q)

418 {

419 void \*p;

420

421 acquire(&q->lock);

422 while((p = q->ptr) == 0)

423 sleep(q, &q->lock);

424 q->ptr = 0;

425 release(&q->lock);

426 return p;

427 }

The fact that recv holds q->lock prevents send from trying to wake it up be- tween recv’s check of q->ptr and its call to sleep. We need sleep to atomically re- lease q->lock and put the receiving process to sleep.

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

## Code: Sleep and wakeup

Let’s look at the implementation of sleep (2874) and wakeup (2953). 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 wait channel and marks it as RUNNABLE. Callers of sleep and wakeup can use any mutually convenient

sleep+code sleep+code wakeup+code SLEEPING+code sched+code RUNNABLE+code

number as the channel. Xv6 often uses the address of a kernel data structure involved in the waiting.

Sleep (2874) begins with a few sanity checks: there must be a current process (2878) and sleep must have been passed a lock (2881-2882). Then sleep acquires pt- able.lock (2891). 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 to 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 en- tirely (2890). For example, wait (2964) calls sleep with &ptable.lock.

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 (2895-2898). At some point later, a process will call wakeup(chan). Wakeup (2964) acquires pt- able.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 (2953) 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.

Xv6 code always calls wakeup while holding the lock that guards the sleep condi- tion; in the example above that lock is q->lock. Strictly speaking it is sufficient if wakeup always follows the acquire (that is, one could call wakeup after the release). Why do the locking rules for sleep and wakeup ensure a sleeping process won’t miss a wakeup it needs? The sleeping process holds either the lock on the condition or the ptable.lock or both from a point before it checks the condition to a point after it is marked as sleeping. If a concurrent thread causes the condition to be true, that thread must either hold the lock on the condition before the sleeping thread acquired it, or after the sleeping thread released it in sleep. If before, the sleeping thread must have seen the new condition value, and decided to sleep anyway, so it doesn’t matter if it misses the wakeup. If after, then the earliest the waker could acquire the lock on the condition is after sleep acquires ptable.lock, so that wakeup’s acquisition of pt- able.lock must wait until sleep has completely finished putting the sleeper to sleep. Then wakeup will see the sleeping process and wake it up (unless something else wakes it up first).

It is sometimes the case that multiple processes are sleeping on the same channel; for example, more than one process reading 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

ptable.lock+code wakeup+code ptable.lock+code ptable.lock+code wakeup1+code wakeup+code SLEEPING+code chan+code RUNNABLE+code

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.

No harm is done if two uses of sleep/wakeup accidentally choose the same chan- nel: 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 indirec- tion (callers need not know which 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 use sleep and wakeup to synchronize readers and writers. One is in the IDE driver: a process adds a disk request to a queue and then calls sleep. The IDE interrupt handler uses wakeup to alert the process that its request has completed.

A 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 ex- amine the file descriptor support surrounding pipes, but let’s look now at the imple- mentations 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]. The counts do not wrap. This convention lets the implementation distin- guish 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 (6830) begins by acquiring the pipe’s lock, which protects the counts, the data, and their associated invariants. Piperead (6851) then tries to acquire the lock too, but cannot. It spins in acquire (1574) 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 (6844). During this loop, it could happen that the buffer fills (6836). 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 enters its crit- ical section: it finds that p->nread != p->nwrite (6856) (pipewrite went to sleep be- cause p->nwrite == p->nread+PIPESIZE (6836)) so it falls through to the for loop, copies data out of the pipe (6863-6867), and increments nread by the number of bytes copied. That many bytes are now available for writing, so piperead calls wakeup (6868) 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.

pipewrite+code piperead+code struct pipe+code RUNNABLE+code

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, exit, and kill

Sleep and wakeup can be used for many kinds of waiting. An interesting example, seen in Chapter 0, is the wait system call that a parent process uses to wait for a child to exit. 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. If the parent exits before the child, the init process adopts the child and waits for it, so that every child has a parent to clean up after it.

An implementation challenge is the possibility of races between parent and child wait and exit, as well as exit and exit. 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 have exited, it calls sleep to wait for one of them to exit (2707) and scans again. Here, the lock being released in sleep is ptable.lock, the special case we saw above.

Exit acquires ptable.lock and then wakes up any process sleeping on a wait channel equal to the current process’s parent proc (2651); if there is such a process, it will be the parent in wait. This may look premature, since exit has not marked the current process as a ZOMBIE yet, but it is safe: although wakeup may cause the parent to run, 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 exit has set its state to ZOMBIE (2663). Before exit yields the processor, it reparents all of the exiting process’s children, passing them to the initproc (2653-2660). Finally, exit calls sched to relinquish the CPU.

If the parent process was sleeping in wait, the scheduler will eventually run it. The call to sleep returns holding ptable.lock; wait rescans the process table and finds the exited child with state == ZOMBIE. (2657). It records the child’s pid and then cleans up the struct proc, freeing the memory associated with the process (2687-2694).

The child process could have done most of the cleanup during exit, but it is im- portant 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 proce- dure runs on its own stack rather than on the stack of the thread that called sched.

While exit allows a process to terminate itself, kill (2975) lets one process re- quest that another be terminated. It would be too complex for kill to directly de- stroy the victim process, since the victim might be executing on another CPU or sleep- ing while midway through updating kernel data structures. To address these chal- lenges, kill does very little: it just sets the victim’s p->killed and, if it is sleeping,

wait+code ZOMBIE+code struct proc+code exit+code sched+code

1. >kstack+code p->pgdir+code swtch+code

wakes it up. Eventually the victim will enter or leave the kernel, at which point code in trap will call exit if p->killed is set. If the victim is running in user space, it will soon enter the kernel by making a system call or because the timer (or some other device) interrupts.

If the victim process is in sleep, the call to wakeup will cause the victim process to return from sleep. This is potentially dangerous because the condition being wait- ing for may not be true. However, xv6 calls to sleep are always wrapped in a while loop that re-tests the condition after sleep returns. Some calls to sleep also test p-

>killed in the loop, and abandon the current activity if it is set. This is only done when such abandonment would be correct. For example, the pipe read and write code (6837) returns if the killed flag is set; eventually the code will return back to trap, which will again check the flag and exit.

Some xv6 sleep loops do not check p->killed because the code is in the middle of a multi-step system call that should be atomic. The IDE driver (4379) is an example: it does not check p->killed because a disk operation may be one of a set of writes that are all needed in order for the file system to be left in a correct state. To avoid the complication of cleaning up after a partial operation, xv6 delays the killing of a process that is in the IDE driver until some point later when it is easy to kill the pro- cess (e.g., when the complete file system operation has completed and the process is about to return to user space).

## Real world

The xv6 scheduler implements a simple scheduling policy, which runs each pro- cess 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 process. These policies can become complex quickly because there are often competing goals: for example, the operating might also want to guaran- tee fairness and high throughput. In addition, complex policies may lead to unintend- ed 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 prevent the high-priority process from making progress. A long convoy can form when many high-priority processes are waiting for a low-pri- ority process that acquires a shared lock; once a convoy has formed it can persist for long 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 ‘‘lost wakeups’’ prob- lem 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 lost wakeups. The Linux kernel’s sleep uses an

round robin priority inversion convoys

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 re- fer 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 fla- vor: 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 par- ticular channel, and it might be the case that many processes are waiting for that par- ticular channel. The operating system will schedule 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 incre- ments 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 ‘‘lost wakeup’’ problem: there is an explicit count of the number of wakeups that have occurred. The count also avoids the spurious wake- up and thundering herd problems.

Terminating processes and cleaning them up introduces much complexity in xv6. In most operating systems it is even more complex, because, for example, the victim process may be deep inside the kernel sleeping, and unwinding its stack requires much careful programming. Many operating systems unwind the stack using explicit mecha- nisms for exception handling, such as longjmp. Furthermore, there are other events that can cause a sleeping process to be woken up, even though the event it is waiting for has not happened yet. For example, when a Unix process is sleeping, another pro- cess may send a signal to it. In this case, the process will return from the interrupt- ed system call with the value -1 and with the error code set to EINTR. The application can check for these values and decide what to do. Xv6 doesn’t support signals and this complexity doesn’t arise.

Xv6’s support for kill is not entirely satisfactory: there are sleep loops which probably should check for p->killed. A related problem is that, even for sleep loops that check p->killed, there is a race between sleep and kill; the latter may set p-

>killed and try to wake up the victim just after the victim’s loop checks p->killed

but before it calls sleep. If this problem occurs, the victim won’t notice the p-

>killed until the condition it is waiting for occurs. This may be quite a bit later (e.g., when the IDE driver returns a disk block that the victim is waiting for) or never (e.g., if the victim is waiting from input from the console, but the user doesn’t type any in-

thundering herd signal+code

put).

## Exercises

* 1. Sleep has to check lk != &ptable.lock to avoid a deadlock (2890-2893). Sup- pose the special case were eliminated by replacing

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

with

}

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

Doing this would break sleep. How?

* 1. 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.
  2. 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 re- sult.
  3. Fix the race mentioned above between kill and sleep, so that a kill that oc- curs after the victim’s sleep loop checks p->killed but before it calls sleep results in the victim abandoning the current system call.
  4. Design a plan so that every sleep loop checks p->killed so that, for example, a process that is in the IDE driver can return quickly from the while loop if another kills that process.
  5. Design a plan that uses only one context switch when switching from one user process to another. This plan involves running the scheduler procedure on the kernel stack of the user process, instead of the dedicated scheduler stack. The main challenge is to clean up a user process correctly. Measure the performance benefit of avoiding one context switch.
  6. Modify xv6 to turn off a processor when it is idle and just spinning in the loop in scheduler. (Hint: look at the x86 HLT instruction.)
  7. The lock p->lock protects many invariants, and when looking at a particular piece of xv6 code that is protected by p->lock, it can be difficult to figure out which invariant is being enforced. Design a plan that is more clean by perhaps splitting p-

>lock in several locks.

# Chapter 6 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 as *persistence* so that data is still available after a reboot.

The xv6 file system provides Unix-like files, directories, and pathnames (see Chap- ter 0), and stores its data on an IDE disk for persistence (see Chapter 3). The file sys- tem addresses several challenges:

* The file system needs on-disk data structures to represent the tree of named di- rectories and files, to record the identities of the blocks that hold each file’s con- tent, 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, so the file system code 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.

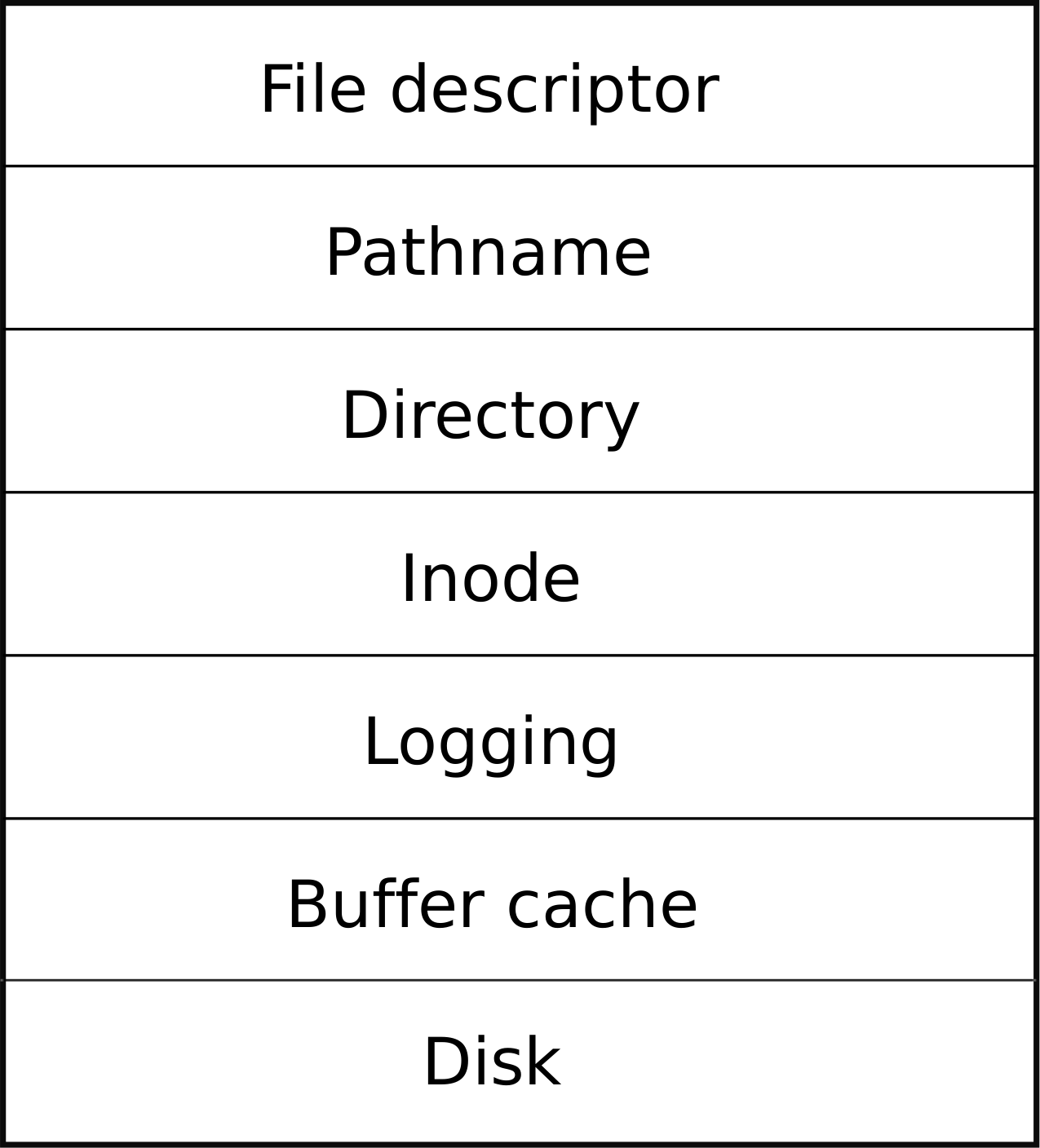
The rest of this chapter explains how xv6 addresses these challenges.

## Overview

The xv6 file system implementation is organized in seven layers, shown in [Figure](#_bookmark21) [6-1](#_bookmark21). The disk layer reads and writes blocks on an IDE hard drive. The buffer cache layer caches disk blocks and synchronizes access to them, making sure that only one kernel process at a time can modify the data stored in any particular block. The log- ging layer allows higher layers to wrap updates to several blocks in a *transaction*, and ensures that the blocks are updated atomically in the face of crashes (i.e., all of them are updated or none). The inode layer provides individual files, each represented as an *inode* with a unique i-number and some blocks holding the file’s data. The directory layer implements each directory as a special kind of inode whose content is a sequence of directory entries, each of which contains a file’s name and i-number. The pathname layer provides hierarchical path names like /usr/rtm/xv6/fs.c, and resolves them with recursive lookup. The file descriptor layer abstracts many Unix resources (e.g., pipes, devices, files, etc.) using the file system interface, simplifying the lives of applica- tion 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 6-2](#_bookmark22). The file system does not use block 0 (it holds the boot sector). Block 1 is called the

persistence crash recovery transaction inode



**Figure 6-1**. Layers of the xv6 file system.

Directory Inode Logging

Buffer cache

*superblock*; it contains metadata about the file system (the file system size in blocks, the number of data blocks, the number of inodes, and the number of blocks in the log). Blocks starting at 2 hold the log. After the log are the inodes, with multiple inodes per block. After those come bitmap blocks tracking which data blocks are in use. The remaining blocks are data blocks; each is either marked free in the bitmap block, or holds content for a file or directory. The superblock is filled in by a separate program, called mfks, which builds an initial file system.

The rest of this chapter discusses each layer, starting with the buffer cache. Look out for situations where well-chosen abstractions at lower layers ease the design of higher ones.

## 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 need 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 *buf* 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 uses a per-buffer sleep-lock to ensure that only one thread at a time uses each buffer (and thus each disk block); bread returns a locked buffer, and brelse releases the lock.

Let’s return to the buffer cache. 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 al- ready 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

superblock mfks+code bread+code bwrite+code buf brelse+code

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| boot | super | log |  | ino | des |  | bit ma | p | data | .... | data |

0 1 2

**Figure 6-2**. Structure of the xv6 file system. The header fs.h (4050) contains constants and data struc- tures describing the exact layout of the file system.

soon.

## Code: Buffer cache

The buffer cache is a doubly-linked list of buffers. The function binit, called by main (1230), initializes the list with the NBUF buffers in the static array buf (4450-4459). All other access to the buffer cache refer to the linked list via bcache.head, not the buf array.

A buffer has two state bits associated with it. B\_VALID indicates that the buffer contains a copy of the block. B\_DIRTY indicates that the buffer content has been mod- ified and needs to be written to the disk.

Bread (4502) calls bget to get a buffer for the given sector (4506). If the buffer needs to be read from disk, bread calls iderw to do that before returning the buffer.

Bget (4466) scans the buffer list for a buffer with the given device and sector num- bers (4472-4480). If there is such a buffer, bget acquires the sleep-lock for the buffer. bget then returns the locked buffer.

If there is no cached 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 buffer that is not locked and not dirty: any such buffer can be used. Bget edits the buffer metadata to record the new device and sector number and ac- quires its sleep-lock. Note that the assignment to flags clears B\_VALID, thus ensuring that bread will read the block data from disk rather than incorrectly using the buffer’s previous contents.

It is important that there is at most one cached buffer per disk sector, to ensure that readers see writes, and because the file system uses locks on buffers for synchro- nization. bget ensures this invariant by holding the bache.lock continuously from the first loop’s check of whether the block is cached through the second loop’s declara- tion that the block is now cached (by setting dev, blockno, and refcnt). This causes the check for a block’s presence and (if not present) the designation of a buffer to hold the block to be atomic.

It is safe for bget to acquire the buffer’s sleep-lock outside of the bcache.lock critical section, since the non-zero b->refcnt prevents the buffer from being re-used for a different disk block. The sleep-lock protects reads and writes of the block’s buffered content, while the bcache.lock protects information about which blocks are cached.

If all the buffers are busy, then too many processes are simultaneously executing file system calls; bget panics. A more graceful response might be to sleep until a

binit+code main+code NBUF+code bcache.head+code B\_VALID+code B\_DIRTY+code bget+code iderw+code bget+code bget+code B\_VALID+code

buffer became free, though there would then be a possibility of deadlock.

Once bread has read the disk (if needed) and returned the buffer to its caller, the caller has exclusive use of the buffer and can read or write the data bytes. If the caller does modify the buffer, it must call bwrite to write the changed data to disk before releasing the buffer. Bwrite (4515) calls iderw to talk to the disk hardware, after setting B\_DIRTY to indicate that iderw should write (rather than read).

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 (4526) releases the sleep-lock and moves the buffer to the front of the linked list (4537-4542). Moving the buffer causes the list to be ordered by how recently the buffers 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 (start- ing 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 buffer by scanning backward (following prev pointers).

## Logging layer

One of the most interesting problems in 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 incon- sistent state. For example, suppose a crash occurs during file truncation (setting the length of a file to zero and freeing its content blocks). Depending on the order of the disk writes, the crash may either leave an inode with a reference to a content block that is marked free, or it may leave an allocated but unreferenced content block.

The latter is relatively benign, but an inode that refers to a freed block is likely to cause serious problems after a reboot. After reboot, the kernel might allocate that block to another file, and now we have two different files pointing unintentionally to the same block. If xv6 supported multiple users, this situation could be a security problem, since the old file’s owner would be able to read and write blocks in the new file, owned by a different user.

Xv6 solves the problem of crashes during file system operations with a simple form 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 all of its writes, it writes a special *commit* record to the disk indicating that 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 the log. The recovery code finishes by erasing the log.

bread+code bwrite+code iderw+code B\_DIRTY+code brelse+code

log commit

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 com- mits, 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, specified in the superblock. It consists of a header block followed by a sequence of updated block copies (‘‘logged blocks’’). The header block contains an array of sector numbers, one for each of the logged blocks, and the count of log blocks. The count in the header block on disk is either zero, indicating that there is no transaction in the log, or non-zero, indicating that the log contains a complete committed transaction with the indicated number 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 with respect to crashes. To allow concurrent execution of file system operations by different processes, the logging system can accumulate the writes of mul- tiple system calls into one transaction. Thus a single commit may involve the writes of multiple complete system calls. To avoid splitting a system call across transactions, the logging system only commits when no file system system calls are underway.

The idea of committing several transactions together is known as *group commit*.

Group commit reduces the number of disk operations because it amortizes the fixed cost of a commit over multiple operations. Group commit also hands the disk system more concurrent writes at the same time, perhaps allowing the disk to write them all during a single disk rotation. Xv6’s IDE driver doesn’t support this kind of *batching*, but xv6’s file system design allows for it.

Xv6 dedicates a fixed amount of space on the disk to hold the log. The total number of blocks written by the system calls in a transaction must fit in that space. This has two consequences. No single system call can be allowed to write more dis- tinct 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 sys- tem call breaks up large writes into multiple smaller writes that fit in the log, and un- link doesn’t cause problems because in practice the xv6 file system uses only one bitmap block. The other consequence of limited log space is that the logging system cannot allow a system call to start unless it is certain that the system call’s writes will fit in the space remaining in the log.

group commit batching write+code unlink+code

## Code: logging

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

begin\_op+code log\_write+code bwrite+code absorption end\_op+code install\_trans+code

begin\_op();

recover\_from\_log+cod

...

bp = bread(...);

bp->data[...] = ...; log\_write(bp);

...

end\_op();

begin\_op (4828) waits until the logging system is not currently committing, and until there is enough unreserved log space to hold the writes from this call. log.outstanding counts the number of system calls that have reserved log space; the total reserved space is log.outstanding times MAXOPBLOCKS. Incrementing log.outstanding both reserves space and prevents a commit from occuring during this system call. The code conservatively assumes that each system call might write up to MAXOPBLOCKS distinct blocks.

log\_write (4922) acts as a proxy for bwrite. It records the block’s sector number in memory, reserving it a slot in the log on disk, and marks the buffer B\_DIRTY to pre- vent the block cache from evicting it. The block must stay in the cache until commit- ted: until then, the cached copy is the only record of the modification; it cannot be written to its place on disk until after commit; and other reads in the same transaction must see the modifications. log\_write notices when a block is written multiple times during a single transaction, and allocates that block the same slot in the log. This op- timization is often called *absorption*. It is common that, for example, the disk block containing inodes of several files is written several times within a transaction. By ab- sorbing several disk writes into one, the file system can save log space and can achieve better performance because only one copy of the disk block must be written to disk.

end\_op (4853) first decrements the count of outstanding system calls. If the count

is now zero, it commits the current transaction by calling commit(). There are four stages in this process. write\_log() (4885) copies each block modified in the transac- tion from the buffer cache to its slot in the log on disk. write\_head() (4804) writes the header block to disk: this is the commit point, and a crash after the write will re- sult in recovery replaying the transaction’s writes from the log. install\_trans (4772) reads each block from the log and writes it to the proper place in the file system. Fi- nally end\_op writes the log header with a count of zero; this has to happen before the next transaction starts writing logged blocks, so that a crash doesn’t result in recovery using one transaction’s header with the subsequent transaction’s logged blocks.

recover\_from\_log (4818) is called from initlog (4756), which is called during boot before the first user process runs. (2865) It reads the log header, and mimics the actions of end\_op if the header indicates that the log contains a committed transac- tion.

initlog+code

filewrite+code

An example use of the log occurs in filewrite (6002). The transaction looks like

this:

begin\_op(); ilock(f->ip);

r = writei(f->ip, ...); iunlock(f->ip); end\_op();

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.

## Code: Block allocator

File and directory content is stored in disk blocks, which must be allocated from a free pool. 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 program mkfs sets the bits corresponding to the boot sector, su- perblock, log blocks, inode blocks, and bitmap blocks.

The block allocator provides two functions: balloc allocates a new disk block, and bfree frees a block. Balloc The loop in balloc at (5022) 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 any one bitmap block at a time.

Bfree (5052) finds the right bitmap block and clears the right bit. Again the exclu- sive use implied by bread and brelse avoids the need for explicit locking.

As with much of the code described in the remainder of this chapter, balloc and

bfree must be called inside a transaction.

## Inode layer

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.

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 (4078). The type field distin- guishes between files, directories, and special files (devices). A type of zero indicates that an on-disk inode is free. The nlink field counts the number of directory entries that refer to this inode, in order to recognize when the on-disk inode and its data

writei+code balloc+code bfree+code inode

struct dinode+code

blocks should be freed. The size field records the number of bytes of content in the file. The addrs array records the block numbers of the disk blocks holding the file’s content.

The kernel keeps the set of active inodes in memory; struct inode (4162) 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.

There are four lock or lock-like mechanisms in xv6’s inode code. icache.lock protects the invariant that an inode is present in the cache at most once, and the in- variant that a cached inode’s ref field counts the number of in-memory pointers to the cached inode. Each in-memory inode has a lock field containing a sleep-lock, which ensures exclusive access to the inode’s fields (such as file length) as well as to the inode’s file or directory content blocks. An inode’s ref, if it is greater than zero, causes the system to maintain the inode in the cache, and not re-use the cache entry for a different inode. Finally, each inode contains a nlink field (on disk and copied in memory if it is cached) that counts the number of directory entries that refer to a file; xv6 won’t free an inode if its link count is greater than zero.

A struct inode pointer returned by iget() is guaranteed to be valid until the corresponding call to iput(); the inode won’t be deleted, and the memory referred to by the pointer won’t be re-used for a different inode. iget() provides non-exclusive access to an inode, so that there can be many pointers to the same inode. Many parts of the file system code depend on this behavior of iget(), both to hold long-term ref- erences to inodes (as open files and current directories) and to prevent races while avoiding deadlock in code that manipulates multiple inodes (such as pathname lookup).

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 acqui- sition of inode pointers from locking helps avoid deadlock in some situations, for ex- ample 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; caching is secondary. If an inode is used frequently, the buffer cache will probably keep it in memory if it isn’t kept by the inode cache. The inode cache is write-through, which means that code that modifies a cached inode must immediately write it to disk with iupdate.

## Code: Inodes

struct inode+code iget+code iput+code ilock+code

To allocate a new inode (for example, when creating a file), xv6 calls ialloc (5204). 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 (5218). 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 (5254) 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. (5263-5267). As iget scans, it records the position of the first empty slot (5268- 5269), which it uses if it needs to allocate a cache entry.

Code must lock the inode using ilock before reading or writing its metadata or content. Ilock (5303) uses a sleep-lock for this purpose. Once ilock has exclusive ac- cess to the inode, it reads the inode from disk (more likely, the buffer cache) if needed. The function iunlock (5331) releases the sleep-lock, which may cause any processes sleeping to be woken up.

Iput (5358) releases a C pointer to an inode by decrementing the reference count (5376). 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 calls itrunc to truncate the file to zero bytes, freeing the data blocks; sets the inode type to 0 (unallocated); and writes the inode to disk (5366).

The locking protocol in iput in the case in which it frees the inode deserves a closer look. One danger is that a concurrent thread might be waiting in ilock to use this inode (e.g. to read a file or list a directory), and won’t be prepared to find the in- ode is not longer allocated. This can’t happen because there is no way for a system call to get a pointer to a cached inode if it has no links to it and ip->ref is one. That one reference is the reference owned by the thread calling iput. It’s true that iput checks that the reference count is one outside of its icache.lock critical section, but at that point the link count is known to be zero, so no thread will try to acquire a new refer- ence. The other main danger is that a concurrent call to ialloc might choose the same inode that iput is freeing. This can only happen after the iupdate writes the disk so that the inode has type zero. This race is benign; the allocating thread will po- litely wait to acquire the inode’s sleep-lock before reading or writing the inode, at which point iput is done with it.

iput() can write to the disk. This means that any system call that uses the file system may write the disk, because the system call may be the last one having a refer- ence to the file. Even calls like read() that appear to be read-only, may end up calling iput(). This, in turn, means that even read-only system calls must be wrapped in transactions if they use the file system.

There is a challenging interaction between iput() and crashes. iput() doesn’t truncate a file immediately when the link count for the file drops to zero, because some process might still hold a reference to the inode in memory: a process might still

ialloc+code balloc+code iget+code iget+code ilock+code ilock+code iunlock+code iput+code itrunc+code iput+code

dinode



...

indirect block

address 128

.....

address 1

indirect

address 12

.....

address 1

size

nlink

minor

major

type

data

data

data

data



...

**Figure 6-3**. The representation of a file on disk.

be reading and writing to the file, because it successfully opened it. But, if a crash hap- pens before the last process closes the file descriptor for the file, then the file will be marked allocated on disk but no directory entry points to it.

File systems handle this case in one of two ways. The simple solution is that on recovery, after reboot, the file system scans the whole file system for files that are marked allocated, but have no directory entry pointing to them. If any such file exists, then it can free those files.

The second solution doesn’t require scanning the file system. In this solution, the file system records on disk (e.g., in the super block) the inode inumber of a file whose link count drops to zero but whose reference count isn’t zero. If the file system re- moves the file when its reference counts reaches 0, then it updates the on-disk list by removing that inode from the list. On recovery, the file system frees any file in the list. Xv6 implements neither solution, which means that inodes may be marked allo- cated on disk, even though they are not in use anymore. This means that over time

xv6 runs the risk that it may run out of disk space.

## Code: Inode content

The on-disk inode structure, struct dinode, contains a size and an array of block numbers (see [Figure 6-3](#_bookmark23)). 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

struct dinode+code NDIRECT+code

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 indi- rect 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 (5410) begins by picking off the easy case: the first NDIRECT blocks are listed in the inode itself (5415-5419). The next NINDIRECT blocks are listed in the indirect block at ip->addrs[NDIRECT]. Bmap reads the indirect block (5426) and then reads a block number from the right position within the block (5427). If the block number exceeds NDIRECT+NINDIRECT, bmap panics; writei contains the check that prevents this from happening (5566).

Bmap allocates blocks as needed. An ip->addrs[] or indirect entry of zero indi- cates that no block is allocated. As bmap encounters zeros, it replaces them with the numbers of fresh blocks, allocated on demand. (5416-5417, 5424-5425).

itrunc frees a file’s blocks, resetting the inode’s size to zero. Itrunc (5456) starts by freeing the direct blocks (5462-5467), then the ones listed in the indirect block (5472- 5475), and finally the indirect block itself (5477-5478).

Bmap makes it easy for readi and writei to get at an inode’s data. Readi (5503) starts by making sure that the offset and count are not beyond the end of the file. Reads that start beyond the end of the file return an error (5514-5515) while reads that start at or cross the end of the file return fewer bytes than requested (5516-5517). The main loop processes each block of the file, copying data from the buffer into dst (5519-5524). writei (5553) 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 (5566-5567); the loop copies data into the buffers instead of out (5572); and if the write has extended the file, writei must update its size (5577-5580).

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 (5488) copies inode metadata into the stat structure, which is exposed to user programs via the stat system call.

## Code: directory layer

A directory is implemented internally much like a file. Its inode has type T\_DIR and its data is a sequence of directory entries. Each entry is a struct dirent (4115), which contains a name and an inode number. The name is at most DIRSIZ (14) char- acters; if shorter, it is terminated by a NUL (0) byte. Directory entries with inode number zero are free.

The function dirlookup (5611) searches a directory for an entry with the given

direct blocks NINDIRECT+code

indirect block BSIZE+code bmap+code readi+code writei+code bmap+code NDIRECT+code NINDIRECT+code

itrunc+code readi+code writei+code writei+code readi+code writei+code readi+code writei+code T\_DEV+code stati+code stat+code T\_DIR+code struct dirent+code DIRSIZ+code dirlookup+code

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 scenar- ios 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 (5652) 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 (5658-5662). The main loop reads directory entries looking for an unallocated entry. When it finds one, it stops the loop early (5622-5623), 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 (5672-5675).

## Code: Path names

Path name lookup involves a succession of calls to dirlookup, one for each path component. Namei (5790) evaluates path 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 general- ized function namex to do the real work.

Namex (5755) starts by deciding where the path evaluation begins. If the path be- gins with a slash, evaluation begins at the root; otherwise, the current directory (5759- 5762). Then it uses skipelem to consider each element of the path in turn (5764). 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 (5765-5769). (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 (5770-5774). Finally, the loop looks for the path element using dirlookup and prepares for the next iteration by setting ip = next (5775-5780). When the loop runs out of path elements, it returns ip.

The procedure namex may take a long time to complete: it could involve several disk operations to read inodes and directory blocks for the directories traversed in the pathname (if they are not in the buffer cache). Xv6 is carefully designed so that if an invocation of namex by one kernel thread is blocked on a disk I/O, another kernel thread looking up a different pathname can proceed concurrently. namex locks each directory in the path separately so that lookups in different directories can proceed in parallel.

This concurrency introduces some challenges. For example, while one kernel thread is looking up a pathname another kernel thread may be changing the directory

iget+code

.+code

..+code dirlink+code dirlookup+code nameiparent+code namex+code skipelem+code ilock+code nameiparent+code namex+code dirlookup+code

tree by unlinking a directory. A potential risk is that a lookup may be searching a di- rectory that has been deleted by another kernel thread and its blocks have been re- used for another directory or file.

Xv6 avoids such races. For example, when executing dirlookup in namex, the lookup thread holds the lock on the directory and dirlookup returns an inode that was obtained using iget. iget increases the reference count of the inode. Only after receiving the inode from dirlookup does namex release the lock on the directory. Now another thread may unlink the inode from the directory but xv6 will not delete the inode yet, because the reference count of the inode is still larger than zero.

Another risk is deadlock. For example, next points to the same inode as ip when looking up ".". Locking next before releasing the lock on ip would result in a deadlock. To avoid this deadlock, namex unlocks the directory before obtaining a lock on next. Here again we see why the separation between iget and ilock is important.

## File descriptor layer

A cool aspect of the Unix interface is that most resources in Unix are represented as files, 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 (4150), 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 (5876) scans the file table for an unreferenced file (f->ref == 0) and returns a new reference; filedup (5902) in- crements the reference count; and fileclose (5914) decrements it. When a file’s refer- ence 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 (5952) 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 (5975-5976, 6015-6016). Pipes have no concept of off-

struct file+code open+code dup+code fork+code ftable+code filealloc+code filedup+code fileclose+code fileread+code filewrite+code filedup+code fileclose+code filestat+code fileread+code filewrite+code stat+code read+code write+code stati+code

set. Recall that the inode functions require the caller to handle locking (5955-5957, 5974- 5977, 6025-6028). 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 sys- tem 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 transac- tions. Sys\_link (6202) begins by fetching its arguments, two strings old and new (6207). Assuming old exists and is not a directory (6211-6214), sys\_link increments its ip-

>nlink count. Then sys\_link calls nameiparent to find the parent directory and final path element of new (6227) and creates a new directory entry pointing at old’s in- ode (6230). The new parent directory must exist and be on the same device as the ex- isting 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 ei- ther 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 (6357) 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 directory, 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 (6367). If the name does exist, create’s be- havior 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 (6371). Otherwise, it is an error (6372-6373). If the name does not al- ready exist, create now allocates a new inode with ialloc (6376). 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 (6389). 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 (6401) 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 (6414). Otherwise, it calls namei (6420). 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

sys\_link+code sys\_unlink+code nameiparent+code sys\_link+code create+code open+code O\_CREATE+code mkdir+code mkdev+code sys\_link+code create+code nameiparent+code dirlookup+code mkdir+code mkdev+code T\_FILE+code ialloc+code

.+code

..+code create+code sys\_link+code sys\_open+code sys\_mkdir+code sys\_mknod+code open+code O\_CREATE+code namei+code sys\_open+code

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 (6432) and then fills in the file (6442-6446). Note that no other process can access the partially initialized file since it is only in the current process’s table.

Chapter 5 examined the implementation of pipes before we even had a file sys- tem. The function sys\_pipe connects that implementation to the file system by pro- viding 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 in- stalls 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 elimi- nate the linked list, instead using a hash table for lookups and a heap for LRU evic- tions. Modern buffer caches are typically integrated with the virtual memory system to support memory-mapped files.

Xv6’s logging system is inefficient. A commit cannot occur concurrently with file system system calls. The system 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 inconsisten- cies. Scavenging can take hours for large file systems, and there are situations where it is not possible to resolve inconsistencies in a way that causes the original system calls to be atomic. Recovery from a log is much faster and causes system calls to be atomic in the face of crashes.

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 dur- ing 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 bal- anced tree of blocks. This is 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 hard- ware 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

sys\_pipe+code fsck+code

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 file 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 im- plementing as much of this logic in software as possible. These software implementa- tions typically allow 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 in- ode blocks used by xv6 would not work well in such environments. Separating disk management from the file system may be the cleanest design, but the complex inter- face between the two has led some systems, like Sun’s ZFS, to combine them.

Xv6’s file system lacks many other features of modern file systems; for example, it lacks support for snapshots and incremental backup.

Modern Unix systems allow many kinds of resources to be accessed with the same system calls as on-disk storage: named pipes, network connections, remotely-ac- cessed network file systems, and monitoring and control interfaces such as /proc. In- stead of xv6’s 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 us- er-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 xv6 recover?
2. Why panic in ialloc? Can xv6 recover?
3. Why doesn’t filealloc panic when it runs out of files? Why is this more common and therefore worth handling?
4. 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?
5. 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?
6. 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?
7. Implement the lseek system call. Supporting lseek will also require that you modify filewrite to fill holes in the file with zero if lseek sets off beyond f->ip-

>size.

1. Add O\_TRUNC and O\_APPEND to open, so that > and >> operators work in the shell.

fileread+code filewrite+code

1. Modify the file system to support symbolic links.

# Chapter 7 Summary

This text introduced the main ideas in operating systems by studying one operating system, xv6, line by line. Some code lines embody the essence of the main ideas (e.g., context switching, user/kernel boundary, locks, etc.) and each line is important; other code lines provide an illustration of how to implement a particular operating system idea and could easily be done in different ways (e.g., a better algorithm for scheduling, better on-disk data structures to represent files, better logging to allow for concurrent transactions, etc.). All the ideas were illustrated in the context of one particular, very successful system call interface, the Unix interface, but those ideas carry over to the design of other operating systems.

# 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. Many of the cur- rent standards are public and you can find documentation for them online.

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 memo- ry 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 sim- ple loop: it consults an address in a register called the *program counter*, reads a ma- chine 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.

program counter

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 original- ly defined by Intel and has become a standard. Several manufacturers produce proces- sors 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 *instruc- tion 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 regis- ter 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 reg- isters, 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 regis- ters %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-ac- cess 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 enor- mous 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 be- tween registers and memory both in access time and in size. Today’s x86 processors typically have three levels of cache. Each core has a small first-level cache with access times relatively close to the processor’s clock rate and a larger second-level cache. Sev- eral cores share an L3 cache. [Figure A-1](#_bookmark26) shows the levels in the memory hierarchy and their access times for an Intel i7 Xeon processor.

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 memo- ry—and not worry about the distinctions between the different levels of the memory hierarchy.

instruction pointer control registers segment registers

## I/O

Processors must communicate with devices as well as memory. The x86 processor

**Intel Core i7 Xeon 5500 at 2.4 GHz**

|  |  |  |
| --- | --- | --- |
| **Memory** | **Access time** | **Size** |
| register | 1 cycle | 64 bytes |
| L1 cache | ~4 cycles | 64 kilobytes |
| L2 cache | ~10 cycles | 4 megabytes |
| L3 cache | ~40-75 cycles | 8 megabytes |
| remote L3 | ~100-300 cycles |  |
| Local DRAM | ~60 nsec |  |
| Remote DRAM | ~100 nsec |  |

**Figure A-1**. Latency numbers for an Intel i7 Xeon system, based on [http://software.intel.com](http://software.intel.com/)

/sites/products/collateral/hpc/vtune/performance\_analysis\_guide.pdf.

provides special in and out instructions that read and write values from device ad- dresses called *I/O ports*. The hardware implementation of these instructions is essen- tially the same as reading and writing memory. Early x86 processors had an extra ad- dress 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 as- signed range of I/O ports. A device’s ports let the software configure the device, exam- ine 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 ad- dresses. 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.

I/O ports memory-mapped

I/O

CPU

Selector

Offset

Logical Address

x GB

0

Physical Address

Page Translation

Linear Address

RAM

Segment Translation

logical address linear address physical address

**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 (Basic In-

put/Output System), 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 sys- tem. 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: instruc- tions 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 load- er comprises two source files, one written in a combination of 16-bit and 32-bit x86 assembly (bootasm.S; (9100)) and one written in C (bootmain.c; (9200)).

## Code: Assembly bootstrap

The first instruction in the boot loader is cli (9112), which disables processor in- terrupts. Interrupts are a way for hardware devices to invoke operating system func- tions 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 3), 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 ad- dress 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 pro- gram 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 regis- ter 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.

boot loader real mode

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](#_bookmark28)). 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 2), it translates linear addresses to physical addresses; otherwise the processor uses linear ad- dresses 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 di- rectly 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 en- abled, 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 (9115-9118).

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 pro- vided 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 (9120- 9136).

boot loader logical address linear address virtual address protected mode

segment descriptor table

gdt+code

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 physi- cal addresses to have many more bits, and (since the 80386) a ‘‘32-bit’’ mode that caus- es 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](#_bookmark29)). 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 2 describes. The boot loader sets up the segment descriptor table gdt (9182- 9185) 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-

Logical Address Linear Address

32

16

protected mode

Selector

Offset

32 20 12



16

Flags

Limit

Base

8

0

**Figure B-2**. Segments in protected mode.

GDT/LDT

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 protect- ed mode, logical addresses map one-to-one to physical addresses.

The boot loader executes an lgdt instruction (9141) to load the processor’s global descriptor table (GDT) register with the value gdtdesc (9187-9189), 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 (9142-9144). 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 (9153), which allows a code segment selector to be specified. The jump continues execution at the next line (9156) 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 reg- isters with SEG\_KDATA (9158-9161). Logical address now map directly to physical ad- dresses. 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 0x7e00 (512 bytes). Essentially any other sec- tion 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 (9168). 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 (9170-9176). 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 (9177-9178). A real boot loader might attempt to print an error message first.

boot loader global descriptor

table gdtdesc+code gdt+code CR0\_PE+code gdt+code SEG\_KDATA+code bootmain+code

## Code: C bootstrap

The C part of the boot loader, bootmain.c (9200), expects to find a copy of the kernel executable on the disk starting at the second sector. The kernel is an ELF for- mat binary, as we have seen in Chapter 2. To get access to the ELF headers, bootmain loads the first 4096 bytes of the ELF binary (9214). It places the in-memory copy at ad- dress 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 (9238) and calls stosb to zero the remainder of the segment (9240). 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. Thus, function call instructions must mention desti- nation addresses that look like 0x801xxxxx; you can see examples in kernel.asm. This address is configured in kernel.ld (9311). 0x80100000 is a relatively high ad- dress, towards the end of the 32-bit address space; Chapter 2 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 as- sumes 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 instruc- tion 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 (1040). Since xv6 hasn’t set up virtual memory yet, xv6’s entry point is the physical address of entry (1044).

## Real world

The boot loader described in this appendix compiles to around 470 bytes of ma- chine code, depending on the optimizations used when compiling the C code. In or- der to fit in that small amount of space, the xv6 boot loader makes a major simplify- ing 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

readseg+code stosb+code

\_start+code entry+code

boot loader to be able to drive a variety of disk and network controllers and under- stand 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 sim- ple 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. Modern PCs avoid many of the above complexities, because they support the Unified Extensible Firmware Interface (UEFI), which allows the PC to 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. The BIOS is really a small operating system embedded in the hardware, which is present after the computer has booted.

## Exercises

1. Due to sector granularity, the call to readseg in the text is equivalent to read- seg((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. 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?
3. It seems potentially dangerous for the boot loader to copy the ELF header to mem- ory at the arbitrary location 0x10000. Why doesn’t it call malloc to obtain the memo- ry it needs?

# Index

., 86, 88

.., 86, 88

/init, 27, 35

\_binary\_initcode\_size, 25

\_binary\_initcode\_start, 25

\_start, 102

absorption, 80

acquire, 54, 57

addl, 26

address space, 20

allocproc, 23

allocuvm, 26, 35–36

alltraps, 42–43

argc, 36

argfd, 45

argint, 45

argptr, 45

argstr, 45

argv, 36

atomic, 54

B\_DIRTY, 47–48, 77–78

B\_VALID, 47–48, 77

balloc, 81, 83

batch, 49

batching, 79

bcache.head, 77

begin\_op, 80

bfree, 81

bget, 77

binit, 77

block, 47

bmap, 85

boot loader, 22, 99–101

bootmain, 101

bread, 76, 78

brelse, 76, 78

BSIZE, 85

buf, 76

busy waiting, 48, 66

bwrite, 76, 78, 80

chan, 66, 69

child process, 8

cli, 46

commit, 78

conditional synchronization, 65

contexts, 62

control registers, 96

convoys, 72

copyout, 36

coroutines, 64

cp->tf, 44

cpu->scheduler, 26, 62–63

CR0\_PE, 101

CR0\_PG, 23

CR\_PSE, 37

crash recovery, 75

create, 88

critical section, 53

current directory, 14

deadlocked, 67

direct blocks, 85

dirlink, 86

dirlookup, 85–86, 88

DIRSIZ, 85

DPL\_USER, 25, 42

driver, 46

dup, 87

ELF format, 35

ELF\_MAGIC, 35

EMBRYO, 23

end\_op, 80

entry, 22–23, 102

entrypgdir, 23

exception, 39

exec, 9–11, 26, 36, 42

exit, 8, 27, 63–64, 71

fetchint, 45

file descriptor, 10

filealloc, 87

fileclose, 87

filedup, 87

fileread, 87, 90

filestat, 87

filewrite, 80, 87, 90

FL\_IF, 25

fork, 8, 10–11, 87

forkret, 24, 26, 64

freerange, 33

fsck, 89

ftable, 87

gdt, 100–101

gdtdesc, 101

getcmd, 10

global descriptor table, 101 group commit, 79

I/O ports, 97

ialloc, 83, 88

IDE\_BSY, 47

IDE\_DRDY, 47

IDE\_IRQ, 47

ideinit, 47

ideintr, 48, 56

idelock, 55–56

iderw, 47–48, 55–56, 77–78

idestart, 48

idewait, 47

idt, 42

idtinit, 46

IF, 42, 46

iget, 82–83, 86

ilock, 82–83, 86

indirect block, 85

initcode, 27

initcode.S, 25–26, 41

initlog, 80

initproc, 26

inituvm, 25

inode, 15, 75, 81

insl, 48

install\_trans, 80

instruction pointer, 96

int, 40–42

interface design, 7

interrupt, 39

interrupt handler, 40

ioapicenable, 47

iput, 82–83

iret, 26, 41, 44

IRQ\_TIMER,, 46

isolation, 17

itrunc, 83, 85

iunlock, 83

kalloc, 34

KERNBASE, 23

kernel, 7, 19

kernel mode, 18, 40

kernel space, 7, 19

kfree, 33

kinit1, 33

kinit2, 33

kmap, 32

kvmalloc, 30, 32

lapicinit, 46

linear address, 99–100

links, 15

loaduvm, 35

lock, 51

log, 78

log\_write, 80

logical address, 99–100

main, 23, 26, 32–33, 42, 47, 77

malloc, 10

mappages, 32

memory-mapped I/O, 97

mfks, 76

microkernel, 19–20

mkdev, 88

mkdir, 88

monolithic kernel, 17, 19

mpmain, 25

multiplexing, 61

mutual exclusion, 53

mycpu, 65

myproc, 65

namei, 25, 35, 88

nameiparent, 86, 88

namex, 86

NBUF, 77

NDIRECT, 84–85

NINDIRECT, 85

O\_CREATE, 88

open, 87–88

p->context, 24, 26, 64

p->cwd, 25

p->kstack, 21, 71

p->name, 25

p->pgdir, 22, 71

p->state, 22

p->sz, 45

p->xxx, 21

page, 29

page directory, 29

page table entries (PTEs), 29 page table pages, 29

panic, 44

parent process, 8

path, 14

persistence, 75

PGROUNDUP, 33

physical address, 20, 99

PHYSTOP, 32–33

pid, 8, 23

pipe, 13

piperead, 70

pipewrite, 70

polling, 48, 66

popal, 26

popcli, 57

popl, 26

printf, 9

priority inversion, 72

privileged instructions, 18

proc->killed, 44

process, 7–8, 20

program counter, 95

protected mode, 100–101

ptable, 56

ptable.lock, 63–64, 69

PTE\_P, 29

PTE\_U, 26, 30, 32

PTE\_W, 30

pushcli, 57

race condition, 52

read, 87

readi, 35, 85

readseg, 102

real mode, 99

recover\_from\_log, 80

recursive locks, 58

release, 54, 57

ret, 26

root, 14

round robin, 72

RUNNABLE, 25, 64, 68–70

sbrk, 10, 34

sched, 62–64, 68, 71

scheduler, 25, 63–64

sector, 47

SEG\_KDATA, 101

SEG\_TSS, 25

SEG\_UCODE, 25

SEG\_UDATA, 25

seginit, 37

segment descriptor table, 100 segment registers, 96

sequence coordination, 65

serializing, 53

setupkvm, 25, 32, 35

shell, 8

signal, 73

skipelem, 86

sleep, 63, 66, 68

sleep-locks, 58

SLEEPING, 68–69

stat, 85, 87

stati, 85, 87

sti, 46

stosb, 102

struct buf, 47

struct context, 62

struct cpu, 65

struct dinode, 81, 84

struct dirent, 85

struct elfhdr, 35

struct file, 87

struct inode, 82

struct pipe, 70

struct proc, 21, 71

struct run, 33

struct spinlock, 54

struct trapframe, 25

superblock, 76

switchuvm, 25, 42, 46, 64

swtch, 25–26, 62–64, 71

SYS\_exec, 26, 44

sys\_exec, 42

sys\_link, 88

sys\_mkdir, 88

sys\_mknod, 88

sys\_open, 88

sys\_pipe, 89

sys\_sleep, 56

sys\_unlink, 88

syscall, 44

system call, 7

T\_DEV, 85

T\_DIR, 85

T\_FILE, 88

T\_SYSCALL, 26, 42, 44

tf->trapno, 44

thread, 21

thundering herd, 73

ticks, 56

tickslock, 56

time-share, 8, 17

transaction, 75

Translation Look-aside Buffer (TLB), 35, 59

trap, 40

trap, 43–44, 46, 48, 62

trapret, 24, 26, 44

tvinit, 42

type cast, 33

unlink, 79

user memory, 20

user mode, 18, 40

user space, 7, 19

userinit, 24–26

ustack, 36

V2P\_WO, 23

vectors[i], 42

virtual address, 20, 100

wait channel, 66

wait, 8–9, 64, 71

wakeup, 46, 56, 66, 68–69

wakeup1, 69

walkpgdir, 32, 35

write, 79, 87

writei, 81, 85

xchg, 54, 57

yield, 62–64

ZOMBIE, 71