

xv6

a simple, Unix-like teaching operating system

Russ Cox
Frans Kaashoek
Robert Morris

xv6-book@pdos.csail.mit.edu

Draft as of September 3, 2014

Contents

0	Operating system interfaces	7
1	Operating system organization	17
2	Page tables	29
3	Traps, interrupts, and drivers	37
4	Locking	49
5	Scheduling	57
6	File system	71
7	Summary	87
A	PC hardware	89
B	The boot loader	93
	Index	99

Foreword and acknowledgements

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

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

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

Chapter 0

Operating system interfaces

interface design
kernel
process
system call
user space
kernd space

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 multiplexes the hardware allowing many programs to share the computer and run (or appear to run) at the same time. Finally, operating systems provide controlled ways for programs to interact so that they can share data or work together.

An operating system provides services to user programs through 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 implementation right. On the other hand, we may be tempted to offer many sophisticated features to applications. The trick in resolving this tension is to design interfaces that rely on a few mechanisms that can be combined to provide much generality.

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

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

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

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

The collection of system calls that a kernel provides is the interface that user programs see. The xv6 kernel provides a subset of the services and system calls that Unix kernels traditionally offer. Figure 0-2 lists all xv6 system calls.

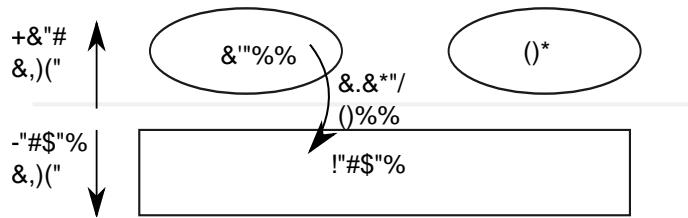


Figure 0-1 A kernel and two user processes.

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

The shell is an ordinary program that reads commands from the user and executes them, and is the primary user interface to traditional Unix-like systems. The fact that the shell is a user program, not part of the kernel, illustrates the power of the system call interface: there is nothing special about the shell. It also means that the shell is easy to replace; 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 implementation can be found at line (8350).

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

Processes and memory

An xv6 process consists of user-space memory (instructions, data, and stack) and per-process state private to the kernel. Xv6 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");
}
```

The exit system call causes the calling process to stop executing and to release re-

System call	Description
fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process ID
sleep(n)	Sleep for n seconds
exec(pname, argv)	Loadable and execute it
sbrk(n)	Grow process's memory by n bytes
open(pname, flags)	Open file; 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 file descriptors 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(pname)	Removeable

Figure 0-2 Xv6 system calls

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

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

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

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

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

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

The exec system call replaces the calling process's memory with a new memory image loaded from a file stored in the file system. The file must have a particular format, which specifies which part of the file holds instructions, which part is data, at which instruction to start, etc. 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");

```

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

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 (8501). The main loop reads the input on the command line using getcmd. Then it calls fork , which creates a copy of the shell process. The parent shell calls wait , while the child process runs the command. For example if the user had typed @echo hello @ at the prompt, runcmd would have been called with @echo hello @ as the argument. runcmd(8406) runs the actual command. For @echo hello @@ would call exec (8426). If exec succeeds then the child will execute instructions from echo instead of runcmd. At some point echo will call exit , which will cause the parent to return from wait in main (8501). You might wonder why fork and exec are not combined in a single call; we will see later that separate calls for creating a process and loading a program is a clever design.

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

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

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 descriptor. For simplicity we often refer to the object a file descriptor refers to as a file. The file descriptor interface abstracts away the difference between files, pipes, and devices, making them all look like streams of bytes.

Internally, the xv6 kernel uses the file descriptor as an index into a per-process table, so that every process has a private space of file descriptors starting at zero. By convention, a process reads from file descriptor 0 (standard input), writes output to file descriptor 1 (standard output), and writes error messages to file descriptor 2 (standard error). As we will see, the shell exploits the convention to implement I/O redirection and pipelines. The shell ensures that it always has three file descriptors open (8507) 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

The 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 error occurs. Like read, write writes data at the current file offset and then advances that offset by the number of bytes written: each write picks up where the previous one left off.

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

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

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

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

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

File descriptors and fork interact to make I/O redirection easy to implement. Fork copies 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 executing 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 does `close(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 (8430). Recall that at this point in the code the shell has already forked the child shell and that `runcmd` will call `exec` to load the new program. Now it should be clear why it is a good idea that `fork` and `exec` are separate calls. This separation allows the shell to `exec` up the child process before the child runs the intended program.

Although `fork` copies the file descriptor table, each underlying file descriptor 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 `helloworld` into a file:

```

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

```

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

File descriptors are a powerful abstraction, because they hide the details of what pipe 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 {
    write(p[1], "hello world\n", 12);
    close(p[0]);
    close(p[1]);
}
```

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

If no data is available a read on a pipe waits for either data to be written or all file descriptors referring to the write end to be closed; in the latter case, read will return 0, just as if the end of a data block 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 manner similar to the above code (8450). The child process creates a pipe to connect the left end of the pipeline with the right end. Then it calls runcmdfor the left end of the pipeline and runcmdfor the right end, and waits for the left and the right ends to finish, by calling wait twice. The right end of the pipeline may be a command that itself

includes a pipe (eg., 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
```

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

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. Xv6 implements directories as a special kind of file. The directories form a tree starting at a special directory called the root. A path like /a/b/c refers to the file or directory namedc inside the directory namedb inside the directory nameda in the root directory /. Paths that do not 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 involved 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 modifies the process's current directory.

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

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

Mknod creates a file in the file system, but the file has no contents. Instead the file's metadata marks it as a device file and records the major and minor device numbers (the two arguments to mknod), which uniquely identify a kernel device. When a pro-

cess later opens the file the kernel diverts read and write system calls to the kernel ^{inode links} 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 inode, can have multiple names, called links . The link system call creates another file system name referring to the same inode as an existing file This fragment creates a new file named both a and b.

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

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

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

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

to the last code sequence leaves the inode and file content accessible asb. Furthermore,

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

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

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

One exception is cd, which is built into the shell (8516) cd must change the current working directory of the shell itself If cd were run as a regular command then the shell would fork a child process, the child process would run cd, 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 descriptors, pipes, and convenient shell syntax for operations on them was a major advance in writing general-purpose reusable programs. The idea sparked a whole culture of software tools that was responsible for much of Unix's power and popularity, and the shell was the basis for the so-called Unix shell language. The Unix system call interface persists today in systems like BSD, Linux, and Mac OS X.

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

The file system abstraction has been a powerful idea, most recently applied to network resource in the form of the World Wide Web. 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 processes onto the underlying hardware, isolate processes from each other, and provide mechanisms for controlled inter-process communication. After studying xv6, you should be able to look at other, more complex operating systems and see the concepts underlying xv6 in those systems as well.

Chapter 1

time-share
isolation
monolithic kernel

Operating system organization

A key requirement for an operating system is to support several activities. For example using the system call interface described in chapter 0 a process can start new processes using fork . The operating system must arrange that these processes can time-share the resource of the computer. For example a process may start more new processes than there are processors in the computer, yet all processes must be able to make some progress. In addition, the operating system must arrange for isolation between the processes. That is, if one process has a bug and fails, it shouldn't impact processes that don't have a dependency on the failed process. Complete isolation, however, is too strong, since it should be possible for processes to interact; for example, it is convenient for users to combine processes to perform complex tasks (eg, by using pipes). Thus, the implementation of an operating system must achieve 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 illustrates this organization by tracing the first process that is created 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 reuse code that xv6 must provide for standard operation. Subsequent chapters will explore each abstraction 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 as arbitrary, with which applications link. In this plan, each application could even have its own library, perhaps tailored to its needs. In this plan, the application can directly interact with the hardware resource and use those resources in the best way for the application (eg, to achieve high performance or predictable performance). Some tiny operating systems for embedded devices or real-time systems are organized in this way.

The downside of this approach is that applications are free to use the library, which means they can also not use it. If they don't use the operating system library, then the operating system cannot enforce time sharing. It must rely on the application to behave properly and, for example periodically give up a process so that another application can run. Such a cooperative time-sharing scheme is maybe OK for a system where all applications trust each other, but doesn't provide strong isolation if applications are mutually distrustful.

To achieve strong isolation a helpful approach is to disallow applications to have direct access to the hardware resource, but instead to abstract the resource into services. For example applications interact with a file system only through open, read, write, and close system calls, instead of reading 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 in Unix applications run as processes using fork, allowing the operating system to save and restore registers on behalf of the application when switching between different processes, so that application doesn't have to be aware of process switching. Furthermore, it allows the operating system to forcefully switch an application out of a process if the application, for example is an end-less loop.

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 and move things around if there is a shortage of memory and provides applications with the convenience of a file system to store their images.

To support controlled interaction between applications, Unix applications can use only file descriptors, instead of to make up some sharing convention of their own (e.g., reserving a piece of physical memory). Unix file descriptors abstract all the sharing details away, hiding from the application if the interaction is happening with the terminal, file system, or pipes, yet allows the operating system to control the interaction. For example if one application fails, it can shut down the communication channel.

As you can see the system call interface in Figure 0-2 is carefully designed to provide programmer convenience but also for the implementation of the interface to enforce 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

To provide strong isolation between the software that uses system calls and the software that implements the system calls, we need a hard boundary between applications and the operating system. If the application makes a mistake we don't want the operating system to fail. Instead the operating system should be able to clean up the application and continue running other applications. This strong isolation means that application shouldn't be able to write over data structures maintained by the operating system, shouldn't be able to overwrite instructions of the operating system, etc.

To provide for such strong isolation processors provide hardware support. For

example the x86 processor like many other processors, has two modes in which the processor executes instructions: kernel mode and user mode. In kernel mode the processor is allowed to execute privileged instructions. For example read and writing to the disk (or any other I/O device) is a privileged instruction. If an application in user mode attempts to execute a privileged instruction, then the processor doesn't execute the instruction, 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 in Chapter 0 illustrates this organization. Applications can execute only user-mode instructions (e.g., adding numbers, etc.) and is said to be running in user space, while the software in kernel mode can execute also privileged instructions and is said to be running in kernel space. The software running in kernel space (or in kernel mode) is called the kernel.

If a user-mode application must read or write to disk, it must transition to the kernel to do so, because the application itself can not execute I/O instructions. Processors 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 processor provides the int 3 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 when transitioning to kernel mode; if the application could decide the kernel entry point, a malicious application could enter the kernel at a point where the validation of arguments etc. is skipped.

Kernel organization

A key design question for an operating system is what part of the operating system should run in kernel mode. A simple answer is that the kernel interface is the system call interface. That is, fork, exec, open, close, read, write, etc. are all kernel calls. This choice means that the complete implementation of the operating system runs in kernel mode. This kernel organization is called a monolithic kernel.

In this organization the complete 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 needs full hardware privilege. Furthermore, it's 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 applications fail too. The computer must reboot to start again.

To reduce the risk of mistakes in the kernel, OS designers can make the lines of code that run in kernel mode small. Most of the operating system doesn't need access

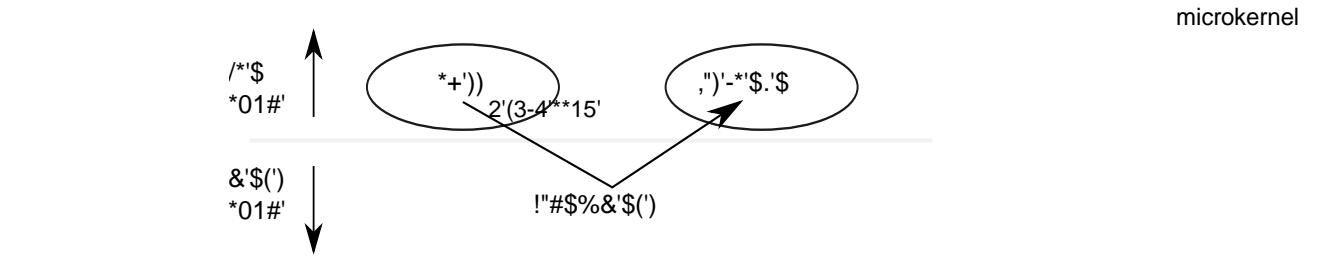


Figure 1-1 A mkernd with a ble system server

to privileged instructions, and can thus run as ordinary user-level applications, with which applications interact with through message. This kernel organization is called a microkernel .

microkernel
process

Figure 1-1 illustrates this microkernel design. In the figure the ble system runs as a user-level application. Operating systems that services that run as ordinary user programs are called servers. To allow application to interact with the ble server the kernel provides a minimal mechanism to send message from one user-mode application to another. For example if an application like the shell wants to read or write a file it sends a message to the ble server and waits for a response.

In a microkernel the kernel interface consists of a few low-level functions for starting applications, performing I/O, sending message to applications, etc. This organization allows the kernel to be implemented with a few lines of code since it doesn't do much, as most functionality of the operating system is implemented by user-level servers.

In the real-world one can find both monolithic kernels and microkernels. For example Linux is mostly implemented as a monolithic kernel although some OS functions run as user-level servers (eg, the windowing system). Xv6 is implemented as a monolithic kernel following most Unix operating systems. 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 functions, 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 processes memory, CPU ble descriptors, etc. It also prevents a process from wrecking the kernel itself (i.e., from preventing the kernel to enforce isolation). The kernel must implement the process abstraction with care because a buggy or malicious application may trick the kernel or hardware in doing something bad (eg., circumventing enforced isolation). The mechanisms used by the kernel to implement processes include user/kernel mode flag address spaces, and time slicing of threads, which this subsection provides an overview of.

To be able to enforce isolation, a process is an abstraction that provides the illusion to a program that it has its own abstract machine. A process provides a program

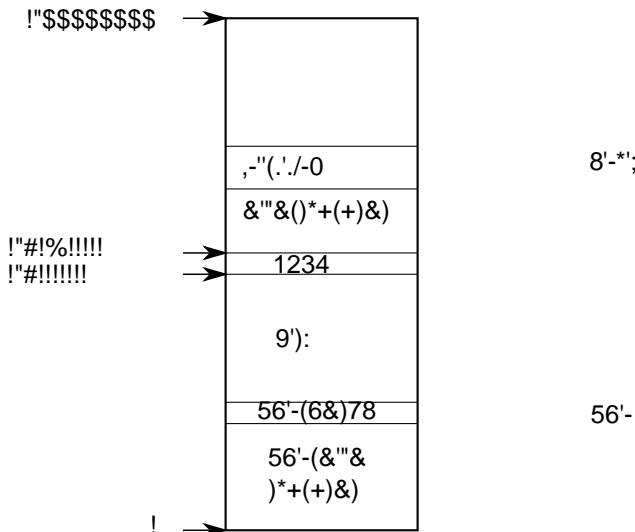


Figure 1-2 Layout of a virtual address space

with what appears 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).

address space
 virtual address
 physical address
 user memory
 struct proc+code
 $p \rightarrow xxx + code$
 thread

Xv6 maintains a separate page table for each process that defines that process's address space. As illustrated in Figure 1-2, 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 (for malloc) that the process can expand as needed.

Each process's address space maps the kernel's instructions and data as well as the user program's memory. When a process invokes a system call, the system call executes in the kernel mappings of the process's address space. This arrangement exists so that the kernel's system call code can directly refer to user memory. In order to leave room for user memory to grow, xv6's address spaces map the kernel at high addresses, starting at 0x80100000.

The xv6 kernel maintains many pieces of state for each process, which it gathers into a struct proc (2353). 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 $p \rightarrow xxx$ to refer to elements of the proc structure.

Each process has a thread of execution (or thread for short) that executes the process's instructions. A thread can be suspended and later resumed. To switch transparently 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.

`p->kstack+code`
`p->state+code`
`p->pgdir+code`
boot loader
entry+code
KERNBASE+code

When a process makes a system call, the process switches to the kernel stack, raises the hardware privilege level and starts executing the kernel instructions that implement 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 resumes executing user instructions just after the system call instruction. A process's thread can block the kernel to wait for I/O, and resume where it left off when the I/O has finished.

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

`p->pgdir` holds the process's page table in the format that the x86 hardware expects. 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 look how the kernel creates the first address space (for itself), how the kernel creates and starts the `Pr$` process, and the `Pr$` system call that that process makes. By tracing these operations we see in detail how xv6 provides strong isolation for processes. The `Pr$` step in providing strong isolation 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 (1040). The x86 paging hardware is not enabled when the kernel starts; virtual addresses map directly to physical addresses.

The boot loader loads the xv6 kernel into memory at physical address 0x100000. The reason it doesn't load the kernel at 0x80100000 where the kernel expects to find its instructions and data, is that there may not be any physical memory at such a high address on a small machine. The reason it places the kernel at 0x100000 rather than 0x0 is because the address range 0xa0000:0x100000 contains I/O devices.

To allow the rest of the kernel to run, entry sets up a page table that maps virtual addresses starting at 0x80000000 (called KERNBASE) to physical addresses starting at 0x0 (see Figure 1-2). 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(1311). We look at the details of page tables in Chapter 2, but the short story is that entry 0 maps virtual addresses

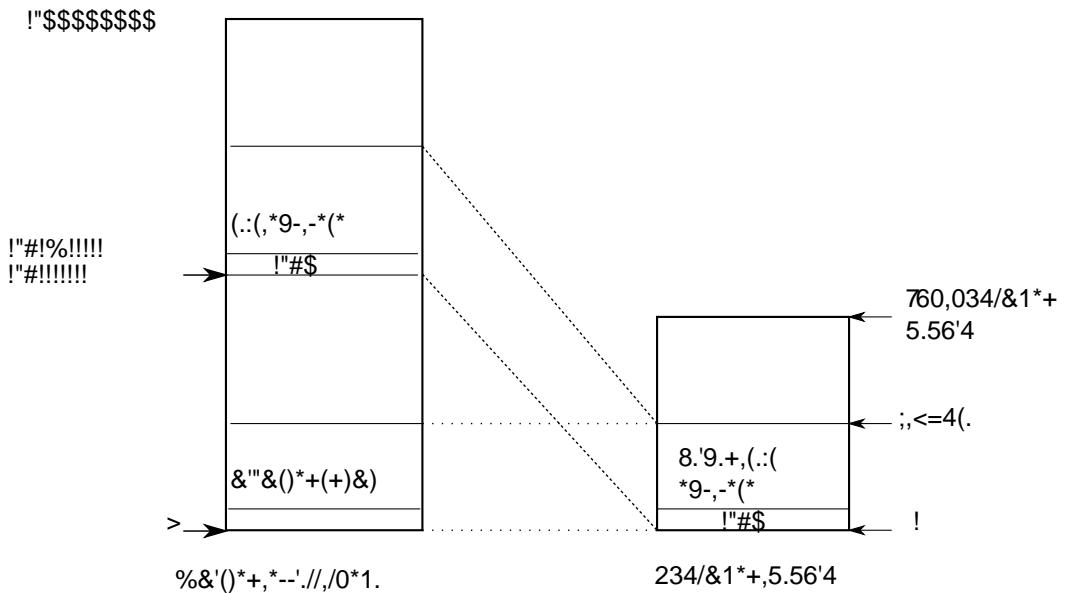


Figure 1-3 Layout of a virtual address space

0:0x400000 to physical addresses 0:0x400000. This mapping is required as long as entry is executing at low address, but will eventually be removed.

Entry 512 maps virtual addresses KERNBASE:KERNBASE+0x400000 physical addresses 0:0x400000. This entry will be used by the kernel after entry has finished; it maps the high virtual address at which the kernel expects to find its instructions and data to the low physical address 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 register %cr3. The paging hardware must know the physical address of entrypgdir, because it doesn't know how to translate virtual address yet; it doesn't have a page table yet. The symbol entrypgdir refers to an address in high memory, and the macro V2P_WO(220) subtracts KERNBASE in order to find the physical address. To enable the paging hardware xv6 sets the flag CR0_PG in the control register %cr0.

The processor is still executing instructions at low address after paging is enabled which works since entrypgdir maps low address. If xv6 had omitted entry 0 from entrypgdir, the computer would have crashed when trying to execute the instruction 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 (1054). All symbols have high address, 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 assembly would otherwise generate a PC-relative direct jump, which would execute the low-memory version of main. Main cannot return, since there is no return PC on the stack. Now the kernel is running in high address in the function main (1217).

entry+code
entrypgdir+code
V2P_WO+code
CR0_PG+code
main+code
main+code
main+code

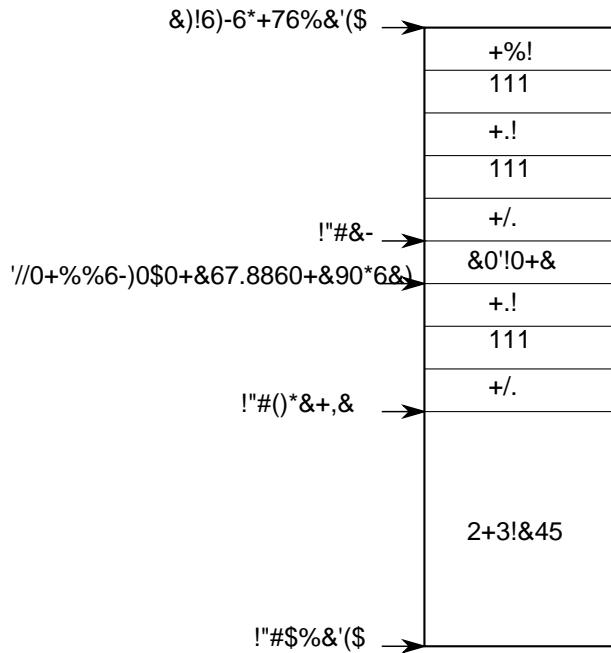


Figure 1-4 A new kernel stack.

Code: creating the `Pr$` process

Now the `kernd` runs within its own address space we look at how the `kernd` creates user-level processes and ensures strong isolation between the `kernd` and user-level processes, and between processes themselves.

After main initializes several devices and subsystems, it creates the `Pr$` process by calling `userinit` (1239). `Userinit`'s `Pr$`ation is to call `allocproc`. The job of `allocproc` (2455) is to allocate a `kt` (a `struct proc`) in the `process` table and to initialize the parts of the `process`'s state required for its `kernd` thread to execute. `Allocproc` is called for each new process, while `userinit` is called only for the very `Pr$` process. `Allocproc` scans the `proc` table for a slot with state `UNUSED` (2461-2463). When it finds an unused slot, `allocproc` sets the state to `EMBRYO`, marks it as used and gives the process a unique `pid` (2451-2469). Next, it tries to allocate a `kernd` stack for the process's `kernd` thread. If the memory allocation fails, `allocproc` changes the state back to `UNUSED` and returns zero to signal failure.

allocproc+code
EMBRYO+code
pid+code
forkret+code
trapret+code
p->context+code
forkret+code

Now `allocproc` must set up the new process's `kernd` stack. `allocproc` is written so that it can be used by `fork` as well as when creating the `Pr$` process. `allocproc` sets up the new process with a specially prepared `kernd` stack and set of `kernd` registers that cause it to return to user space when it `Pr$` runs. The layout of the prepared `kernd` stack will be as shown in Figure 1-4. `allocproc` does part of this work by setting up return program counter values that will cause the new process's `kernd` thread to `Pr$` execute in `forkret` and then in `trapret` (2486-2491). The `kernd` thread will start executing with register contents copied from `p->context`. Thus setting `p->context->eip` to `forkret` will cause the `kernd` thread to execute at the start of `forkret` (2783). This function will return to whatever address is at the bottom of the

stack. The context switch code (2958) sets the stack pointer to point just beyond the end of p->context . allocproc places p->context on the stack, and puts a pointer to trapret just above it; that is where forkret will return. trapret restores user registers from values stored at the top of the kernel stack and jumps into the process(3277). This setup is the same for ordinary fork and for creating the Pr\$ 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 be there if the process had entered the kernel via an interrupt (2514-2520) so that the ordinary code for returning from the kernel back to the process's user code will work. These values are a struct trapframe which stores the user registers. Now the new process's kernel stack is completely prepared as shown in Figure 1-4.

The Pr\$ process is going to execute a small program (initcode.S ; (8200)). The process needs physical memory in which to store this program, the program needs to be copied to that memory and the process needs a page table that refers to that memory.

userinit calls setupkvm(1837) to create a page table for the process with (at Prst) 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 Figure 1-2.

The initial contents of the Pr\$ process's memory are the compiled form of initcode.S; as part of the kernel build process, the linker embeds that binary in the kernel and defines two special symbols, _binary_initcode_start and _binary_initcode_size, 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 binary to that page(1903).

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 privilege level DPL_USER (i.e., user mode not kernel mode), and similarly %ds%es and %ss use SEG_UDATA with privilege DPL_USER. The %eflags FL_IF bit is set to allow hardware interrupts; we will reexamine this in Chapter 3.

The stack pointer %esp is set to the process's target 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 name in detail in Chapter 6.

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

Code: Running the Pr\$ process

Now that the `Pr$` process's state is prepared it is time to run it. After main calls `userinit`, `mpmaincalls` scheduler to start running processes (1267). Scheduler (2708) looks for a process with `p->state` set to `RUNNABLE` and here'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 (1868). 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 interrupts 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` (2958) to perform a context switch to the target process's kernel thread. `swtch` saves the current registers and loads the saved registers of the target kernel thread (`proc->context`) into the x86 hardware registers, including the stack pointer and instruction pointer. The current context is not a process but rather a special per-cpu schedule context, so scheduler tells `swtch` to save the current hardware registers in per-cpu storage (`cpu->scheduler`) rather than in any process's kernel thread context. We'll examine `swtch` in more detail in Chapter 5. The final `ret` instruction (2977) pops the target process's `%eip` from the stack, finishing the context switch. Now the process is running on the kernel stack of `processp`.

`Allocproc` sets `initproc`'s `>context->eip` to `forkret`, so the `ret` starts executing `forkret`. On the `Pr$` invocation (that is this one), `forkret` (2783) runs initialization functions that cannot be run from `main` because they must be run in the context of a regular process with its own kernel stack. Then, `forkret` returns. `Allocproc` arranged that the top word on the stack after `p->context` is popped off would be `trapret`, so now `trapret` begins executing with `%esp` set to `p->tf`. `Trapret` (3277) uses `pop` instructions to restore registers from the trap frame (0602) just as `swtch` did with the kernel context: `popal` restores the general registers, then the `popl` instructions restore `%gs`, `%fs`, `%es` and `%ds`. The `addl` skips over the two `hldstrapno` and `errcode`. Finally, the `iret` instruction pops `%cs`, `%eip`, `%flags`, `%esp` and `%ss` from the stack. The contents of the trap frame have been transferred to the CPU state, so the process continues at the `%eip` specified in the trap frame. For `initproc`, that means virtual address zero, the `Pr$` 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 process's paging hardware translates them into physical addresses. `allocuvvm` sets up the process's page table so that virtual address zero refers to the physical memory allocated for this process, and sets a flag `PTE_U` that tells the paging hardware to allow user code to access that memory. The fact that `userinit` (2514) sets up the low bits of `%cst` 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 sensitive hardware registers such as `%cr3`. So the process is constrained to using only its own memory.

The `Pr$` system call: exec

Now we have seen how the kernel provides strong isolation for processes, but see how a user-level process can enter back into the kernel to ask for services that it cannot perform itself.

The purpose 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.

```
exec+code
SYS_exec+code
T_SYSCALL+code
exec+code
exit+code
/init+code
initcode+code
/init+code
```

Initcode.S (8208) begins by pushing three values on the stack: \$argv, \$init, and \$0 and then sets %eax to SYS_exec and executes int T_SYSCALL. It is asking the kernel to run the exec system call. If all goes well, exec never returns: it starts running the program named by \$init, which is a pointer to the NUL-terminated string /init (8221-8223). If the exec fails and does return, initcode bops calling the exit system call, which definitely should not return (8215-8219).

The arguments to the exec system call are \$init and \$argv. The final zero makes this hand-written system call look like the ordinary system calls, as we will see in Chapter 3. As before this setup avoids special-casing the first process (in this case, its first system call), and instead reuse code that xv6 must provide for standard operation.

Chapter 2 will cover the implementation of exec in detail, but at a high level it will replace initcode with the /init binary, loaded out of the file system. Now initcode (8200) is done, and the process will run /init instead. Init (8310) creates a new console device file if needed and then opens it as file descriptors 0, 1, and 2. Then it loops, starting a console shell, handles orphaned zombies until the shell exits, and repeats. The system is up.

Real world

Most operating systems have adopted the process concept, and most processes look similar to xv6. A real operating system would bind 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.

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

Exercises

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

Chapter 2

Page tables

page table entries
(PTEs)
page
page directory
page table pages
PTE_P+code
PTE_W+code

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 processes. The level of indirection provided by page tables is also a source for 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.

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

Each PTE contains flag bits that tell the paging hardware how the associated virtual address is allowed to be used. PTE_R indicates whether the PTE is present: if it is not set, a reference to the page causes a fault (i.e. is not allowed). PTE_W controls

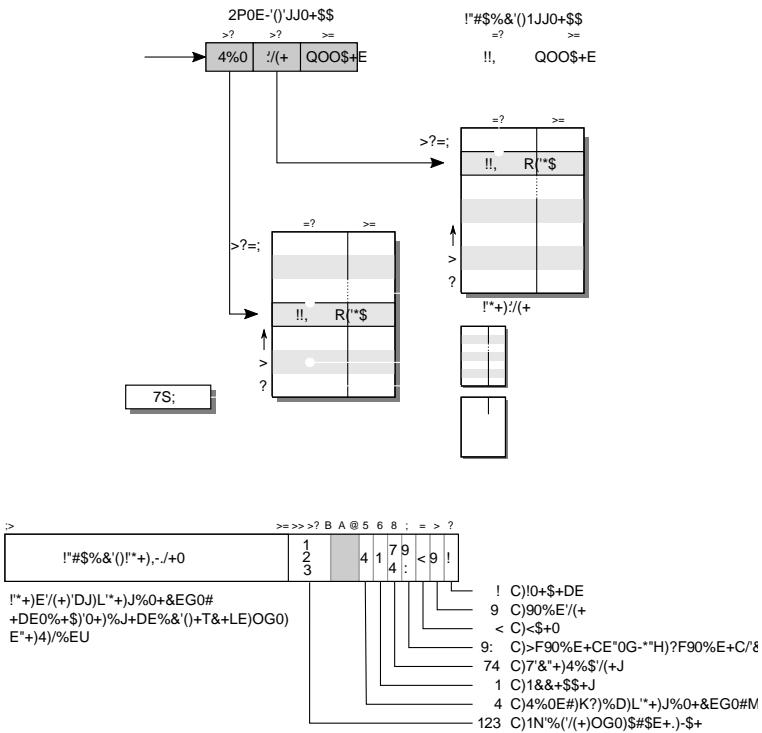


Figure 2-1 x86 page table hardware.

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 shows how it all works. The bag and all other page hardware related structures are defined in mmu.h(0700).

PTE_U+code
kvmalloc+code

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 (1857) because kernel has a more elaborate plan for describing process address spaces.

Each process has a separate page table and xv6 tells the page table hardware to switch page tables when xv6 switches between processes. As shown in Figure 2-2, a process's user memory starts at virtual address zero and can grow up to KERNBASE, following a process to address up to 2 GB from memory. The file memlayout.h (0200) declares the constants for xv6's memory layout and macros to convert virtual to physical

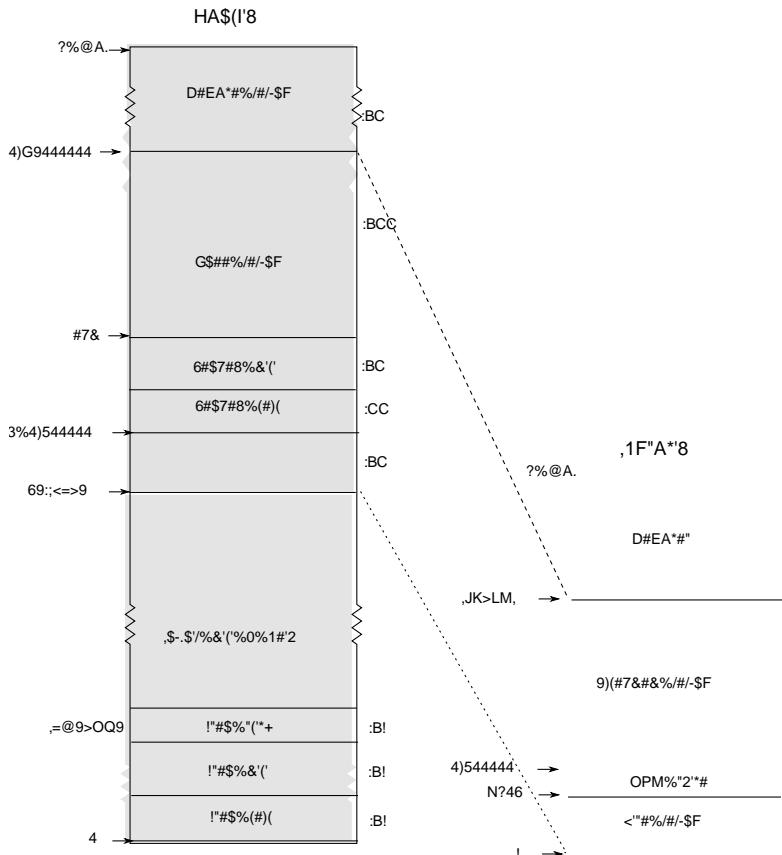


Figure 2-2 Layout of a virtual address space and the physical address space.

addresses.

PTE_U+code

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_UPTE_Ward PTE_PFlags in these PTEs. Most processes do not use the entire user address space; xv6 leaves PTE_Pclear in unused PTEs. Different processes' page tables translate user addresses to different pages of physical memory so that each process has private user memory.

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

PTE_U+code
main+code
kvmalloc+code
setupkvm+code
mappages+code
kmap+code
PHYSTOP+code
mappages+code
walkpgdir+code
walkpgdir+code
PHYSTOP+code

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

Code: creating an address space

main calls kvmalloc (1857) 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(1837). 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(1828) 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(1779) installs mappings into a page table for a range of virtual addresses to a corresponding range of physical addresses. It does this separately for each virtual address in the range, at page intervals. For each virtual address to be mapped, mappages calls walkpgdir to find the address of the PTE for that address. It then initializes the PTE to hold the relevant physical page number, the desired permissions (PTE_W and/or PTE_U and PTE_R), and marks the PTE as valid (1791).

walkpgdir (1754) mimics the actions of the x86 paging hardware as it looks up the PTE for a virtual address (see Figure 2-1). walkpgdir uses the upper 10 bits of the virtual address to find the page directory entry (1759). If the page directory entry isn't present, then the required page table page hasn't yet been allocated; if the alloc argument is set, walkpgdir allocates it and puts its physical address in the page directory. Finally it uses the next 10 bits of the virtual address to find the address of the PTE in the page table page (1772).

Physical memory allocation

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

xv6 uses the physical memory between the end of the kernel and PHYSTOP for run-time allocation. It allocates and frees whole 4096-byte pages at a time. It keeps track of which pages are free by maintaining 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.

```
strut run+code
main+code
kinit1+code
kinit2+code
PHYSTOP+code
freerange+code
kfree+code
PGROUNDUP+code
type cast
kalloc+code
```

Code: Physical memory allocator

The allocator's data structure is a free list of physical memory pages that are available for allocation. Each free page's list element is a struct run (3014). Where does the allocator get the memory to hold that data structure? It stores 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 (3018-3022). 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 (3030). The reason for having two calls is that for much of main one cannot use locks or memory above 4 megabyte. The call to kinit1 sets up for lock-less allocation in the first 4 megabyte, 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 240 megabytes (PHYSTOP) of physical memory and uses all the memory between the end of the kernel and PHYSTOP as the initial pool of free memory. 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 PGROUND to ensure that it frees only aligned physical addresses. 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 (3065) begins by setting every byte in the memory being freed to the value 1. This will cause code that uses memory after freeing it (uses dangling references) to read garbage instead of the old valid contents; hopefully that will cause such code to break faster. Then kfree casts v to a pointer to struct run, records the old start of the free list in r->next, and sets the free list equal to r. kalloc removes and returns the first element in the free list.

User part of an address space

Figure 2-3 shows the layout of the user memory of an executing process in xv6.

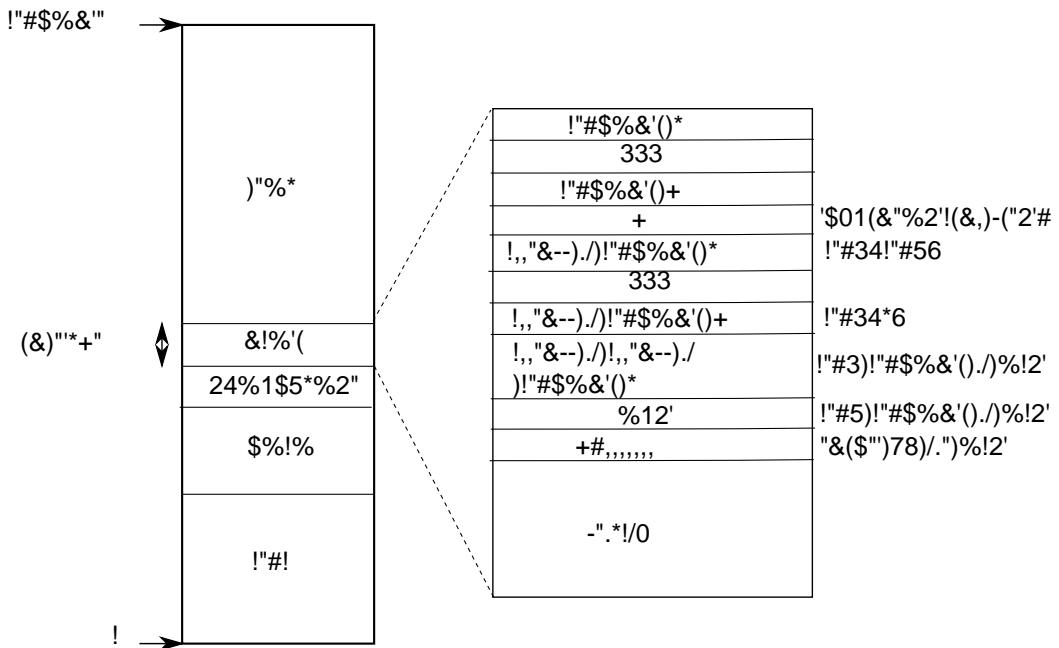


Figure 2-3 Memory layout of a user process with its initial stack.

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

sbrk+code
namei+code
ELF format
struct elfhdr+code
ELF_MAGIC+code

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 (6310) opens the named binary path using namei (6321), 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` (0955), followed by a sequence of program sections, `struct proghdr` (0974). Each `proghdr` describes a section of the application that must be loaded into memory; xv6 programs have only one program section header, but other systems might have separate sections for instructions and data.

The first step is a quick check that the file probably contains an ELF binary. An ELF binary starts with the four-byte magic number `0x7F`, `OEOLOF` or `ELF_MAGIC` (0952). 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(6334), allocates memory for each ELF segment with allocuvvm (6346) and loads each segment into memory with loaduvvm(6348). allocuvvm checks that the virtual address requested is below KERNBASE. loaduvvm(1918) uses walkpgdir to find the physical address of the allocated memory at which to write each page of the ELF segment and ready 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 0x00000054vaddr 0x00000000 paddr 0x00000000 align 2**2
    filesz 0x000008c0 memsz 0x000008cc flags rwx
```

The program section header 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 allocuvvm 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 function that exec uses to copy arguments to the stack will notice that the destination page is not accessible and will return D1.

During the preparation of the new memory image if exec detects an error like an invalid program segment it jumps to the label bad, frees the new image and returns D1. 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 D1 to it. The only error case in exec happens during the creation of the image. Once the image is complete, exec can install the new image(6394) and free the old one (6395). Finally, exec returns 0.

Real world

Like most operating systems, xv6 uses the paging hardware for memory protection and mapping. Most operating systems make far more sophisticated use of paging than xv6; for example xv6 lacks demand paging from disk, copy-on-write fork, shared memory lazily-allocated pages, and automatically extending stacks. The x86 supports address translation using segmentation (see Appendix B), but xv6 uses segments only

```
setupkvm+code
allocuvvm+code
loaduvvm+code
loaduvvm+code
walkpgdir+code
readit+code
/init+code
allocuvvm+code
exec+code
ustack+code
argv+code
argc+code
copyout+code
```

for the common trick of implementing per-CPU variables such as `proc` that are at a^{`seginit+code`}
~~boxed~~ address but have different values on different CPUs (see `seginit`). Implementations of per-CPU (or per-thread) storage on non-segmented architectures would dedicate a register to holding a pointer to the per-CPU data area, but the x86 has so few general registers that the extra effort required to use segmentation is worthwhile.

On machines with lots of memory it might make sense to use the x86's 4 MiB 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 Kbytes of memory giving it a 4 MiB physical page is wasteful. Large 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(1311). The array initialization sets two of the 1024 PEs, at indices zero and $512 \ll \text{KERNBASE} \gg \text{PDXSHIFT}$ leaving the other PEs zero. Xv6 sets the `PTE_PSE` bit in these two PEs 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 240 MB. On the x86, there are at least three common algorithms: the first is to probe the physical address space looking for regions that behave like memory preserving the values written to them; the second is to read the number of kilobytes of memory out of a known 16-bit location in the PC's non-volatile RAM; and the third is to look in BIOS memory for a memory layout table left as part of the multiprocessor tables. Reading the memory layout table is complicated.

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

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 `entrypgdir`?
3. Modify xv6 so that the pages for the kernel are shared among processes, which reduces memory consumption.
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.

Chapter 3

Traps, interrupts, and drivers

When running a process, a CPU executes the normal process loop: read an instruction, 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., referencing a virtual address for which there is no PTE) or a user program asking the kernel for a service with a system call. There are three main challenges in handling these events: 1) the kernel must arrange that a process switches from user mode to kernel mode (and back); 2) the kernel and devices must coordinate their parallel activities; and 3) the kernel must understand the interface of the devices well. Addressing these 3 challenges requires detailed understanding of hardware and careful programming and can result in opaque kernel code. This chapter explains how xv6 addresses these three challenges.

System calls, exceptions and interrupts

With a system call a user program can ask for an operating system service as we saw at the end of the last chapter. The term exception refers to an illegal program action that generates an interrupt. Examples of illegal programs actions include divide by zero, attempt to access memory for a PTE that is not present and so on. The term interrupt refers to a signal generated by a hardware device indicating that it needs attention of the operating system. For example a disk 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 to disk interrupts, the kernel must be involved if only to force uncooperative processes to yield the processor.

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

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

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

int+code interrupt handler traps	kernd mode user mode
--	-------------------------

The basic plan is as follows. An interrupt stops the normal process loop and starts executing a new sequence called an interrupt handler. Before starting the interrupt handler, the process 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 process should switch from user mode to kernel mode and back.

A word on terminology. Although the official x86 term is interrupt, xv6 refers to all of these abstractions, largely because it was the term used by the PDP11/40 and therefore is the conventional Unix term. This chapter uses the terms trap and interrupt interchangeably but it is important to remember that traps are caused by the current process running on a processor (e.g., the process makes a system call and as a result generates a trap), and interrupts are caused by devices and may or not be related to the currently running process. For example a disk may generate an interrupt when it is done retrieving a block for one process, but at the time of the interrupt some other process may be running. This property of interrupts makes thinking about interrupts more difficult than thinking about traps, because interrupts happen concurrently with other activities. Both rely, however on the same hardware mechanism to transfer control between user and kernel mode securely which we will discuss next.

X86 protection

The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege). In practice most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode respectively. The current privilege level with which the x86 executes instructions is stored in %cr2 register in the CPU.

On the x86, interrupt handlers are stored 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 interrupt descriptor from the IDT, where n is the argument of int.
- ¥ Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- ¥ Save %esp and %ss in a CPU-internal register, but only if the target segment selector \oplus CPL < CP.
- ¥ Load %ss and %esp from a task segment descriptor.
- ¥ Push %ss.

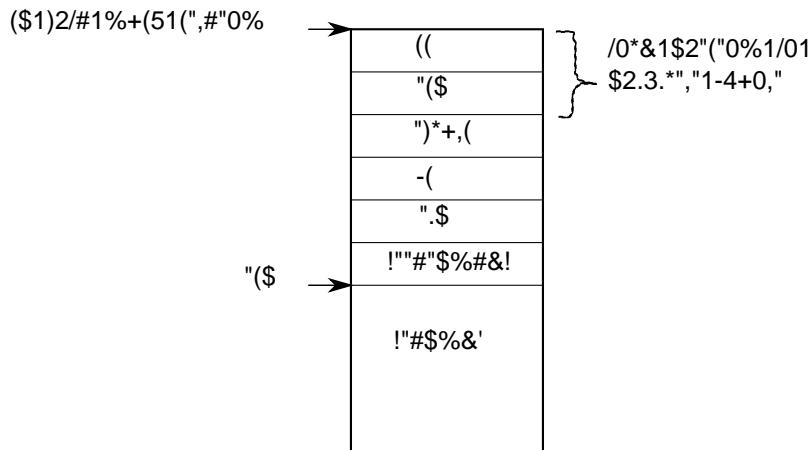


Figure 3-1 Kernel stack after an int instruction.

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

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

Figure 3-1 shows the stack after an int instruction completes and there was a privilege-level change (the privilege level in the descriptor is lower than CPL). If the int instruction didn't require a privilege-level change, the x86 would save %ss and %esp. After both cases, %eip is pointing to the address specified in the descriptor table, and the instruction at that address is the next instruction to be executed and the first instruction of the handler for int n. It is job of the operating system to implement these handlers, and below we will see what xv6 does.

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

Code: The Prst system call

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

(8213) The process pushes the arguments for an exec call on the process 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 (3600). We need to arrange that the int instruction switches the process from user mode to kernel mode that the kernel invokes the right kernel function (i.e., sys_exec), and that the kernel can retrieve the arguments for sys_exec. The next few subsections describes how xv6 arranges this for system calls, and then we will discover that we can reuse the same code for interrupts and exceptions.

```
exec+code
sys_exec+code
int+code
tvinit
main+code
idt+code
vectors[i]+code
T_SYSCALL+code
FL+code
DPL_USER+code
switchuvvm+code
alltraps+code
```

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 reserved for software exceptions, like divide errors or attempts to access invalid memory addresses. Xv6 maps the 32 hardware interrupts to the range 32-63 and uses interrupt 64 as the system call interrupt.

Tvinit (3317) called from main, sets up the 256 entries in the tableidt. 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 handles 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 type Trap, passing a value of 1 as second argument. Trap gates don't clear the FL flag allowing other interrupts during the system call handler.

The kernel also sets the system call gate privilege to DPL_USER which allows a user program to generate the trap with an explicit int instruction. xv6 doesn't allow processes to raise other interrupts (eg., 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 task segment selector and a new value for %esp. The function switchuvvm (1873) stores the address of the top of the kernel stack of the user process into the task segment descriptor.

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

xv6 uses a Perl script (3200) to generate the entry points that the IDT entries point to. Each entry pushes an error code if the processor didn't push the interrupt number, and then jumps to alltraps .

Alltraps (3254) continues to save processor registers: it pushes %ds, %es, %fs,

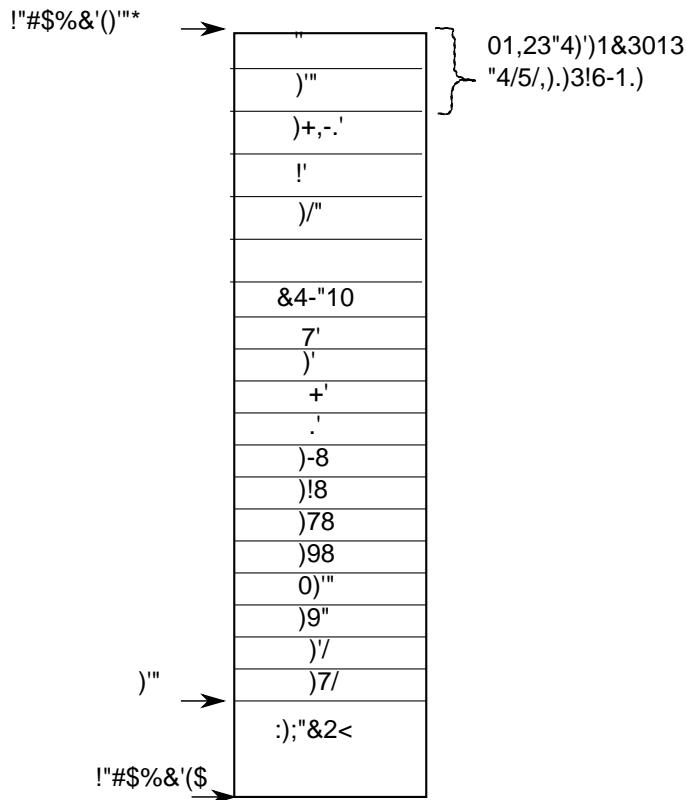


Figure 3-2 The trapframe on the kernel stack

%gs, and the general-purpose registers(3255-3260). The result of this effort is that the kernel stack now contains a struct trapframe (0602) containing the processor registers at the time of the trap (see Figure 3-2). The processor pushes %ss, %esp, %eflags, %cs, and %eip. The processor or the trap vector pushes an error number, and alltraps pushes the rest. The trap frame contains all the information necessary to restore the user mode processor registers when the kernel returns to the current process, so that the processor can continue exactly as it was when the trap started. Recall from Chapter 2, that userinit build a trapframe by hand to achieve this goal (see Figure 1-4).

In the case of the `int $0x80` 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 %flags 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 finish setting up the processor to run kernel C code. The processor set the selectors%cs and %ss before entering the handler; alltraps sets%ds and %es(3263-3265). It sets%fs and %gs to point at the SEG_KCPU data segment(3266-3268).

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` (3271). Then it calls `trap` (3272). After `trap` returns, alltraps pops

the argument of the stack by adding to the stack pointer (3273) and then starts executing 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: popping through the trap frame restores the user mode registers and then iret jumps back into user space.

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

```

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

```

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 (3351) looks at the hardware trap number tf->trapno to decide why it has been called and what needs to be done. If the trap is T_SYSCALL trap calls the system call handler syscall . We'll revisit the two proc->killed checks in Chapter 5.

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

If the trap is not a system call and not a hardware device looking for attention, trap assumes it was caused by incorrect behavior (e.g., divide by zero) as part of the code that was executing before the trap. If the code that caused the trap was a user program, xv6 prints details and then sets cp->killed to remember to clean up the user process. We will look at how xv6 does this cleanup in Chapter 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 (3625). 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(3457) and syscall will invoke the SYS_exec entry of the system call table which corresponds to invoking sys_exec .

Syscall records the return value of the system call function in %eax. When the trap returns to user space it will load the values from cp->tf into the machine registers. Thus, when exec returns, it will return the value that the system call handler returned (3631). 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: binding the system call arguments. The helper functions `argint` and `agptr`, `argstr` retrieve the `n`th system call argument as either an integer pointer, or a string. `argint` uses the user-space%esp register to locate the `n`th argument: %esp points at the return address for the system call stub. The arguments are right above it, at `%esp+4`. Then the `r`th argument is at `%esp+4+4*n`.

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

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

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

The system call implementations (for example `sysproc` and `sysexec`) are typically wrappers: they decode the arguments using `argint`, `agptr`, and `argstr` and then call the real implementations. In chapter 2, `sys_exec` uses these functions to get at its arguments.

Code: Interrupts

Devices on the motherboard can generate interrupts, and xv6 must setup the hardware to handle these interrupts. Without device support xv6 wouldn't be usable; a user couldn't type on the keyboard while the system couldn't store data on disk, etc. Fortunately adding interrupts and support for simple devices doesn't require much additional complexity. As we will see, interrupts can use the same code as for 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 attention (e.g., the user has typed a character on the keyboard). We must program the device to generate an interrupt, and arrange that a CPU receives the interrupt.

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

while not swamping the processor with handling interrupts.

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

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

To also work correctly on uniprocessors, Xv6 programs the programmable interrupt controller (PIC) (7432). Each PIC can handle a maximum of 8 interrupts (i.e., devices) and multiplex them on the interrupt pin of the processor. To allow for more than 8 devices, PCs can be cascaded and typically boards have at least two. Using `inb` and `outb` instructions Xv6 programs the master to generate IRQ 0 through 7 and the slave to generate IRQ 8 through 16. Initially xv6 programs the RC to mask all interrupts. The code in `timer.c` sets timer 1 and enables the timer interrupt on the PIC (8074). This description omits some of the details of programming the RC. These details of the RC (and the IOAPIC and LAPIC) are not important to this text but the interested reader can consult the manuals for each device which are referenced in the source files.

On multiprocessors, xv6 must program the IOAPIC, and the LAPIC on each processor. The IO APIC has a table and the processor can program entries in the table through memory-mapped I/O, instead of using `inb` and `outb` instructions. During initialization, xv6 programs to map interrupt 0 to IRQ 0, and so on, but disables them all. Specific devices enable particular interrupts and say to which processor the interrupt should be routed. For example xv6 routes keyboard interrupts to processor 0 (8016). Xv6 routes disk interrupts to the highest numbered processor on the system, as we will see below.

The timer chip is inside the LAPIC, so that each processor can receive timer interrupts independently. Xv6 sets it up in `lalicinit` (7151). The key line is the one that programs the timer (7164). This line tells the LAPIC to periodically generate an interrupt at IRQ_TIMER which is IRQ 0. Line (7193) 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 (8912) and the other processors (1126). The schedule on each processor enables interrupts (2714). To control that certain code fragments are not interrupted, xv6 disables interrupts during these code fragments (e.g., see `switchuvvm` (1873)).

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

programmable interrupt controller (PIC)
`inb+code`
`outb+code`
`timer.c+code`
`lalicinit+code`
`IRQ_TIMER,+code`
`IF+code`
`cli+code`
`sti+code`
`switchuvvm+code`
`idtinit+code`

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

Trap when it's called for a time interrupt, does just two things: increment the ticks variable(3367) and call wakeup The latter, as we will see in Chapter 5, may cause the interrupt to return in a different process.

trap+code
wakeup+code
driver
block
sector
bu"er
strud buf+code
B_VALID+code
B_DIRTY+code
B_BUSY+code
ideinit+code
main+code
piconable+code
ioapicenable+code
IDE_IRQ+code

Drivers

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

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

Code: Disk driver

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

The dsk driver represents disk sectors with a data structure called a buffer , struct buf (3750) Each bu"er represents the contents of one sector on a particular disk device. The dev and sector fields give the device and sector number and the data field is an in-memory copy of the disk sector.

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

The kernel initializes the dsk driver at boot time by calling ideinit (4151) from main (1234). Ideinit calls piconable and ioapicenable to enable the IDE_IRQ interrupt (4156-4157). The call to piconable enables the interrupt on a uniprocessor ioapicenable enables the interrupt on a multiprocessor but only on the last CPU (hcpu-1): on a two-processor system, CPU 1 handles dsk interrupts.

```

ideinit+code
idewait+code
IDE_BSY+code
IDE_DRDY+code
iderw+code
B_DIRTY+code
B_VALID+code
iderw+code
polling
busy waiting
iderw+code
idestart+code
idestart+code
iderw+code
trap+code
ideintr+code
insl+code

```

Next, ideinit probes the dsk hardware. It begins by calling idewait (4158) to wait for the dsk to be able to accept commands. A PC motherboard presents the status bits of the dsk hardware on I/O port 0x1f7. Idewait (4133) polls the status bits until the busy bit (IDE_BSY) is clear and the ready bit (IDE_DRDY) is set.

Now that the dsk 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 dsk is ready (4160-4167). If not, ideinit assumes the dsk is absent.

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

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

Iderw (4254) adds the buffer b to the end of the queue (4267-4271). If the buffer is at the front of the queue, iderw must send it to the dsk hardware by calling idestart (4224-4226); otherwise the buffer will be started once the buffers ahead of it are taken care of.

Idestart (4175) issues either a read or a write for the buffer's device and sector, according to the flag. If the operation is a write, idestart must supply the data now (4189) and the interrupt will signal that the data has been written to dsk. 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 knowledge about the IDE device and writes the right values at the right ports. If any of these out-of-scope statements is wrong, the IDE will do something differently than what we want. Getting these details right is one reason why writing device drivers is challenging.

Having added the request to the queue and started it if necessary, iderw must wait for the result. As discussed above, polling does not make efficient use of the CPU. Instead, iderw sleeps, waiting for the interrupt handler to record in the buffer's flags that the operation is done (4278-4279). While this process is sleeping, xv6 will schedule other processes to keep the CPU busy.

Eventually the dsk will finish its operation and trigger an interrupt. trap will call ideintr to handle it (3374). Ideintr (4202) consults the first buffer in the queue to poll out which operation was happening. If the buffer was being read and the dsk controller has data waiting, ideintr reads the data into the buffer with insl (4215).

4217) Now the buffer is ready: ideintr sets B_VALID and wakes up any process sleeping on the buffer (4219-4222). Finally, ideintr must pass the next waiting buffer to the dsk (4224-4226).

B_VALID+code
B_DIRTY+code
batch

Real world

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

Actual device drivers are far more complex than the dsk driver in this chapter, but the basic ideas are the same: typically devices are slower than CPU so the hardware uses interrupts to notify the operating system of status changes. Modern disk controllers typically accept 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 system often reordered the request queue themselves.

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

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

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

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

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

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

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

Exercises

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

Chapter 4

Locking

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

The rest of this 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 a line of code in xv6, you must be asking yourself is there another process that could change the intended behavior of the line (eg, because another process is also executing that line or another line of code that modifies a shared variable) and what would happen if an interrupt handler ran. In both cases, you must keep in mind that a single C statement can be several machine instructions and thus another process or an interrupt may muck around in the middle of the C statement. You cannot assume that lines of code on the page are executed sequentially nor can you assume that a single C statement will execute atomically. Concurrency makes reasoning about the correctness much more difficult.

Race conditions

As an example on why we need locks, consider several processors sharing a single disk, such as the IDE disk in xv6. The disk driver maintains a linked list of the outstanding disk requests (4121) and processes may add new requests to the list concurrently (4254). If there were no concurrent requests, you might implement the linked list as follows:

```

1  struct list {
2      int data;
3      struct list *next;
4  };
5
6  struct list *list = 0;
7
8  void
9  insert(int data)
10 {
11     struct list *l;

```

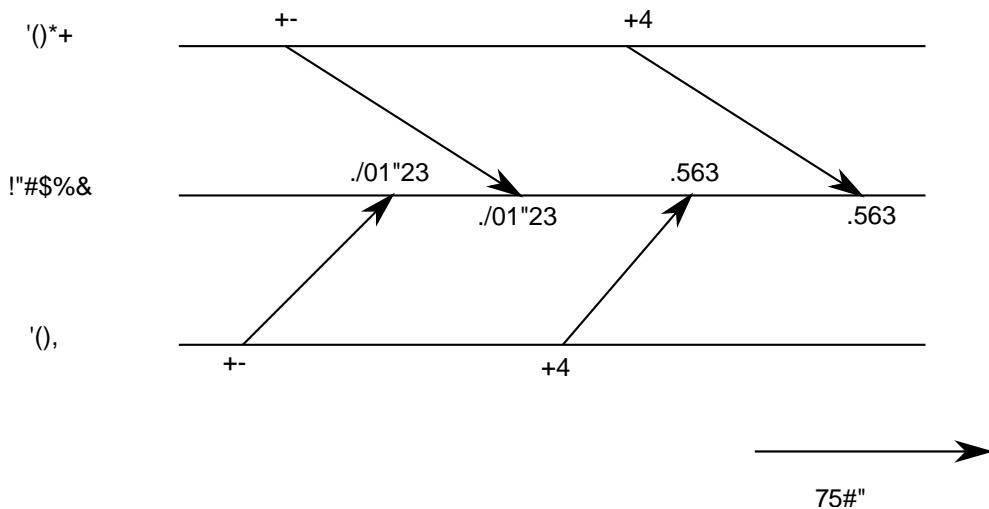


Figure 4-1 Example race

```

12
13     l = malloc(sizeof *l);
14     l->data = data;
15     l->next = l->list;
16     l = t = l;
17 }
```

race condition

Proving this implementation correct is a typical exercise in a data structures and algorithms class. Even though this implementation can be proved correct, it isn't at least not on a multiprocessor. If two different CPUs execute `insert` at the same time, it could happen that both execute line 15 before either executes 16 (see Figure 4-1). If this happens, there will now be two list nodes with `next` set to the former value of `list`. When the two assignments to `list` happen at line 16, the second one will overwrite the first; the node involved in the first assignment will be lost. This kind of problem is called a race condition. The problem with races is that they depend on the exact timing of the two CPUs involved and how their memory operations are ordered by the memory system, and are consequently difficult to reproduce. For example, adding print statements while debugging `insert` might change the timing of the execution enough to make the race disappear.

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

```

6     struct list *list = 0;
7     struct lock listlock;
8
9     void
10    insert(int data)
11    {
12        struct list *l;
```

```

12     acquire(&listlock);
13     l = malloc(sizeof *l);
14     l->data = data;
15     l->next = list;
16     list = l;
17     release(&listlock);
18 }
```

struct
spinlock+code
acquire+code
atomic

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

Code: Locks

Xv6 represents a lock as a struct `spinlock` (1501). The critical field in the structure is locked, a word that is zero when the lock is available and non-zero when it is held. Logically xv6 should acquire a lock by executing code like

```

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

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

To execute those two lines atomically, xv6 relies on a special 386 hardware instruction, xchg (0569). In one atomic operation, xchg swaps a word in memory with the contents of a register. The function acquire (1574) repeats this xchg instruction in a loop; each iteration reads lk->locked and atomically sets it to 1 (1583). If the lock is held, lk->locked will already be 1, so the xchg returns 1 and the loop continues. If the xchg returns 0, however, acquire has successfully acquired the lock. lk->locked was 0 and is now 1, so the loop can stop. Once the lock is acquired, acquire records, for debugging the CPU and task trace that acquired the lock. When a process acquires a lock and forgets to release it, this information can help to identify the culprit. These debugging fields are protected by the lock and must only be edited while holding the lock.

xchg+code
acquire+code
release+code
recursive locks
idlerw+code

The function release (1602) is the opposite of acquire: it deletes the debugging fields and then releases the lock.

Modularity and recursive locks

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

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

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

Code: Using locks

Xv6 is carefully programmed with locks to avoid race conditions. A simple example is in the IDE driver (4100). As mentioned in the beginning of the chapter, iderw (4254) has a queue of disk requests and processes may add new requests to the list concurrently (4269). To protect this list and other invariants in the driver, iderw acquires the

idelock (4265) and release at the end of the function. Exercise 1 explores how to trigger the race condition that we saw at the beginning of the chapter by moving the acquire to after the queue manipulation. It is worthwhile to try the exercise because it will make clear that it is not that easy to trigger the race, suggesting that it is difficult to find race-condition bugs. It is not unlikely that xv6 has some races.

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

The rules above say when locks are necessary but say nothing about when locks are unnecessary and it is important for efficiency not to lock too much, because locks reduce parallelism. If efficiency wasn't important, then one could use a uniprocessor computer and not worry at all about locks. For protecting kernel data structures, it would suffice to create a single lock that must be acquired on entering the kernel and released on exiting the kernel. Many uniprocessor operating systems have been converted to run on multiprocessors using this approach, sometimes called a "big kernel lock," but the approach sacrifices true concurrency: only one CPU can execute in the kernel at a time. If the kernel does any heavy computation, it would be more efficient to use a larger set of more fine-grained locks, so that the kernel could execute on multiple CPUs simultaneously.

Ultimately, the choice of lock granularity is an exercise in parallel programming. Xv6 uses a few coarse data-structure specific locks; for example xv6 uses a single lock protecting the process table and its invariants, which are described in Chapter 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 proceed in parallel. However it complicates operations that have invariants over the whole process table, since they might have to take out several locks. Hopefully the examples of xv6 will help convey how to use locks.

Lock ordering

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

Because xv6 uses coarse-grained locks and xv6 is simple, xv6 has few lock-order chains. The longest chain is only two deep. For example, ideintr holds the ide lock while calling wakeup which acquires the ptable lock. There are a number of other examples involving sleep and wakeup. These orderings come about because sleep and wakeup have a complicated invariant, as discussed in Chapter 5. In the file system there are a number of examples of chains of two because the file system must, for example, acquire a lock on a directory and the lock on a file in that directory to unlink a file from its parent directory correctly. Xv6 always acquires the locks in the order first parent directory and then the file.

```

ideintr+code
wakeup+code
ptable+code
sleep+code
wakeup+code
ticks+code
sys_sleep+code
tickslock+code
iderw+code
idelock+code
ideintr+code
pushcli+code
popcli+code
cli+code
pushcli+code
popcli+code
popcli+code
cli+code
sti+code
acquire+code
xchg+code
release+code
popcli+code
xchg+code
ideintr+code

```

Interrupt handlers

Xv6 uses locks to protect interrupt handlers running on one CPU from non-interrupt code accessing the same data on another CPU. For example, the timer interrupt handler (3364) increments ticks but another CPU might be in sys_sleep at the same time, using the variable (3723). The lock tickslock synchronizes access by the two CPUs to the single variable.

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

To avoid this situation, if a lock is used by an interrupt handler, a process must never hold that lock with interrupts enabled. Xv6 is more conservative: it never holds any lock with interrupts enabled. It uses pushcli (1655) and popcli (1666) to manage a stack of “disabled interrupts” operations (cli is the x86 instruction that disables interrupts). Acquire calls pushcli before trying to acquire a lock (1576) and release calls popcli after releasing the lock (1621). Pushcli (1655) and popcli (1666) are more than just wrappers around cli and sti: they are counted so that it takes two calls to popcli to undo two calls to pushcli; this way, if code acquires two different locks, interrupts will not be reenabled until both locks have been released.

It is important that acquire call pushcli before the xchg that might acquire the lock (1583). 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 (1583).

The interaction between interrupt handlers and non-interrupt code provides a nice example why recursive locks are problematic. If xv6 used recursive locks (a second acquire on a CPU is allowed if the first acquire happened on that CPU too), then interrupt handlers could run while non-interrupt code is in a critical section. This could create havoc since when the interrupt handler runs, invariants that the handler relies on might be temporarily violated. For example, ideintr (4202) assumes that the linked list with outstanding requests is well-formed. If xv6 would have used recursive

locks, then `ideintr` might run while `iderw` is in the middle of manipulating the linked list, and the linked list will end up in an incorrect state.

`iderw+code`
`release+code`

Memory ordering

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

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

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.

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

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

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

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

Exercises

1. Remove the `xchg` in `acquire`. explain what happens when you run xv6?
2. Move the `acquire` in `idew` to before `sleep` is there a race? ~~why don't you observe it~~ when booting xv6 and ~~run~~ stress? increase critical section with a dummy loop; what do you see now? explain.
3. Setting a bit in a buffer's flags is not an atomic operation: the process makes a copy of flags in a register edits the register and writes it back. Thus it is important that two processes are not writing to flags at the same time. xv6 edits the B_BUS bit only while holding buflock but edits the B_VALID and B_WRITE bits without holding any locks. Why is this safe?

Chapter 5

Scheduling

Any operating system is likely to run with more processes than the computer has processors, and so a plan is needed to time-share the processors among the processes. Ideally 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 and have the operating system multiplex multiple virtual processors on a single physical processor. This chapter explains how xv6 multiplexes a process among several processes.

Multiplexing

Xv6 multiplexes by switching each process 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 system call. Second xv6 periodically forces a switch when a process is executing user instructions. 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? Xv6 uses the standard mechanism of context switching; although the idea is simple, the implementation is some of the most opaque code in the system. Second how to do context switching transparently? Xv6 uses the standard technique of using the timer interrupt handler to drive context switches. Third, many CPUs may be switching among processes concurrently and a locking plan is necessary to avoid races. Fourth, when a process has exited its memory and other resources must be freed but it cannot do all of this itself because (for example) it cannot free its own kernel stack while still using it. Xv6 tries to solve these problems as simply as possible, but nevertheless the resulting code is ticky.

xv6 must provide ways for processes to coordinate among themselves. For example, a parent process may need to wait for one of its children to exit, or a process reading 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

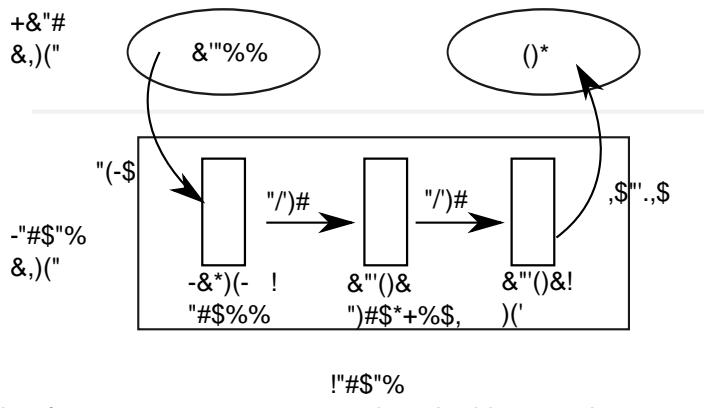


Figure 5-1 Switching from one user process to another. In this example xv6 runs with one CPU (and thus one schedule thread).

As shown in Figure 5-1, to switch between processes, xv6 performs two kinds of context switches at a low level: from a process kernel thread to the current CPU's schedule thread and from the schedule thread to a process kernel thread. xv6 never directly switches from one user-space process to another; this happens by way of a user-kernel transition (system call or interrupt), a context switch to the scheduler, a context switch to a new process kernel thread and a trap return. In this section we'll examine the mechanics of switching between a kernel thread and a schedule thread.

Every xv6 process has its own kernel stack and register set, as we saw in Chapter 2. Each CPU has a separate schedule thread for use when it is executing the scheduler rather than any process kernel thread. Switching from one thread to another involves saving the old thread's CPU registers, and restoring previously-saved registers of the new thread; the fact that %esp and %eip are saved and restored means that the CPU will switch stacks and switch what code it is executing.

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

Instead of following the schedule into swtch, let's instead follow our user process back in. 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 schedule context previously saved in cpu->scheduler (2766).

Swtch (2952) starts by loading its arguments off the stack into the registers %eax and %edx (2959-2960). 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;

swtch+code
contexts
strud context+code
trap+code
yield+code
sched+code
swtch+code
cpu-
>scheduler+code
swtch+code

the convention on the x86 is that these are %ebp, %ebx, %esi, %ebp, and %esp. Swtch pushes the first four explicitly (2963-2966) it saves the last implicitly as the struct context* written to *old (2969). There is one more important register: the program counter %eip was saved by the call instruction that invoked swtch and is on the stack just above %ebp. Having saved the old context, swtch is ready to restore the new one. It moves the pointer to the new context into the stack pointer (2970). 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 (2973-2977). Because swtch has changed the stack pointer, the values restored and the instruction address returned to are the ones from the new context.

In our example, sched called swtch to switch to cpu->scheduler, the per-CPU schedule context. That context had been saved by scheduler's call to swtch (2728). When the swtch we have been tracing returns, it returns not to sched but to scheduler, and its stack pointer points at the current CPU's schedule stack, not initproc's kernel stack.

```

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

```

Code: Scheduling

The last section looked at the low-level details of swtch; now let's take swtch as a given and examine the conventions involved in switching from process to scheduler and back to process. A process that wants to give up the CPU must acquire the process table lock ptable.lock, release any other locks it is holding, update its own state (proc->state), and then call sched Yield (2772) follows this convention, as do sleep and exit, which we will examine later. Sched double-checks those conditions (2757-2762) and then an implication of those conditions: since a lock is held, the CPU should be running with interrupts disabled. Finally, sched calls swtch to save the current context in proc->context and switch to the scheduler context in cpu->scheduler. Swtch returns on the scheduler's stack as though scheduler's swtch had returned (2728). The scheduler continues the for loop, finds a process to run, switches to it, and the cycle repeats.

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

A kernel thread always gives up its process in sched and always switches to the same location in the scheduler, which (almost) always switches to a process in sched. Thus, if one were to print out the line numbers where xv6 switches threads, one would

observe the following simple pattern: (2728) (2766) (2728) (2766), and so on. The procedures in which this stylized switching between two threads happens are sometimes referred to as coroutines ; in this example, sched and scheduler are co-routines of each other.

There is one case when the scheduler \rightarrow swtch to a new process does not end up in sched. We saw this case in Chapter 2: when a new process is first scheduled it begins at forkret (2783). Forkret exists only to honor this convention by releasing the ptable.lock ; otherwise the new process could start at trapret .

Scheduler (2708) runs a simple loop: bind a process to run, run it until it stops, repeat. scheduler holds ptable.lock for most of its actions, but releases the lock (and explicitly enables interrupts) once in each iteration of its outer loop. This is important for the special case in which this CPU is idle (can bind no RUNNABLE process). If an idling schedule 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 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 processes because processes (e.g., the shell) are waiting for I/O; if the schedule left interrupts disabled all the time, the I/O would never arrive.

The schedule 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 switchuvvm, marks the process as RUNNING and then calls swtch to start running it (2722-2728).

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

Maintaining the above invariants is the reason why xv6 acquires ptable.lock in one thread (often in yield) and releases the lock in a different thread (the scheduler thread or another next kernel thread). Once the code has started to modify a running process's state to make it RUNNABLE, it must hold the lock until it has finished restoring the invariants: the earliest correct release point is after scheduler stops using the process's page table and tears 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

```

coroutines
sched+code
scheduler+code
swtch+code
sched+code
forkret+code
ptable.lock+code
scheduler+code
ptable.lock+code
RUNNABLE+code
switchuvvm+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

```

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.

sequence
coordination
conditional
synchronization

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 do 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/consume queue. This queue is similar to the one that feeds commands from processes to the IDE driver (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 optimize too aggressively this implementation would be correct:

```

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

```

Sendloops until the queue is empty ($\text{ptr} == 0$) and then puts the pointer p in the queue. Recvloops until the queue is non-empty and takes the pointer out. When run in different processes send and recv both modify $q->\text{ptr}$, but send only writes the pointer when it is zero and recv only writes the pointer when it is nonzero so no updates 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 receiver's CPU could prod more productive work if there were a way for the receiver to

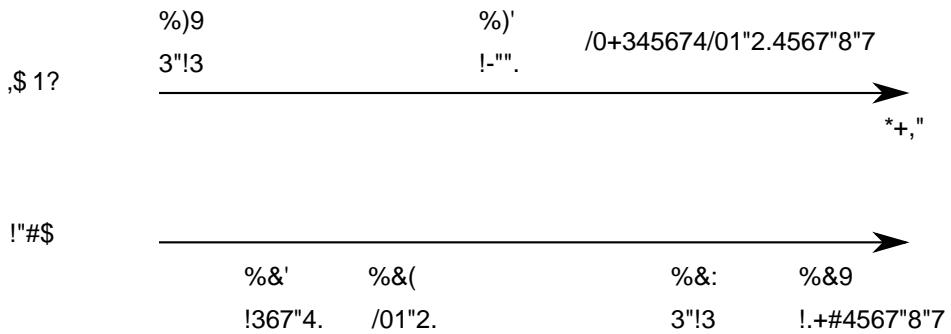


Figure 5-2 Example of a busy-wake problem

yield the CPU and resume only when send delivered a pointer.

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

```

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

Recv now gives up the CPU instead of spinning, which is nice. However it turns out not to be straightforward to design sleep and wakeup with this interface without suffering from what is known as the Oldswake-up problem (see Figure 5-2). Suppose that recv finds that q->ptr == 0 on line 215. While recv is between lines 215 and 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 executing at line 216: it calls sleep and goes to sleep. This causes a problem: recv is asleep waiting for a pointer that has already arrived. The next send will sleep waiting for recv to consume the pointer in the queue at which point the system will be dead-

deadlocked
sleep+code

locked.

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:

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

One might hope that this version of recv would avoid the last wakeup because the 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 fix the preceding scheme by passing 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 receive has finished putting itself to sleep so that the wakeup will find the sleeping receive and wake it up. Once the receive is awake again sleep reacquires the lock before returning. Our new correct scheme is useable as follows:

```
400 struct q {
401     struct spinlock lock;
402     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 between `recv`'s check of `q->ptr` and its call to `sleep`. Of course the receiving process must release `q->lock` while it is sleeping so the sender can wake it up. So we want `sleep` to atomically release `q->lock` and put the receiving process to sleep.

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

Code: Sleep and wakeup

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

`Sleep` (2803) begins with a few sanity checks: there must be a current process (2805) and `sleep` must have been passed a lock (2808-2809). Then `sleep` acquires `ptable.lock` (2818). Now the process going to sleep holds both `ptable.lock` and `lk`. Holding `lk` was necessary in the caller (in the example, `recv`): it ensured that no other process (in the example one running `send`) could start a call `wakeup(chan)`. Now that `sleep` holds `ptable.lock`, it is safe to release `lk`: some other process may start a call to `wakeup(chan)` but `wakeup` will not run until it can acquire `ptable.lock`, so it must wait until `sleep` has finished putting the process to sleep keeping the `wakeup` from missing the sleep.

There is a minor complication: if `lk` is equal to `&ptable.lock`, then `sleep` would deadlock trying to acquire it as `&ptable.lock` and then release it as `lk`. In this case, `sleep` considers the acquire and release to cancel each other out and skips them entirely (2817). For example, `wait` (2653) 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` (2823-2825).

At some point later a process will call wakeup(chan). Wakeup(2853) acquires ptable.lock and calls wakeup1 which does the real work. It is important that wakeup hold the ptable.lock both because it is manipulating process states and because as we just saw, ptable.lock makes sure that sleep and wakeup do not miss each other. Wakeup is a separate function because sometimes the schedule needs to execute a wakeup when it already holds the ptable.lock ; we will see an example of this later. Wakeup(2853) 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 schedule runs, it will see that the process is ready to be run.

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

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

Calls of sleep and wakeup can use any mutually convenient number as the channel; in practice xv6 often uses the address of a kernel data structure involved in the waiting, such as a disk buffer. No harm is done if two uses of sleep/wakeup accidentally choose the same channel: they will see spurious wakeups, but looping as described above will tolerate this problem. Much of the charm of sleep/wakeup is that it is both lightweight (no need to create special data structures to act as sleep channels) and provides a layer of indirection (callers need not know what specific process they are interacting with).

Code: Pipes

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

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

Each pipe is represented by a struct pipe , which contains a lock and a data buffer. The fields nread and nwrite count the number of bytes read from and written

to the bu"er. The bu"er wraps around: the next byte written after buf[PIPE_SIZE-1] is buf[0], but the counts do not wrap. This convention lets the implementation distinguish a full bu"er (`nwrite == nread+PIPE_SIZE`) from an empty bu"er (`nwrite == nread`), but it means that indexing into the bu"er must use `buf[nread % PIPE_SIZE]` instead of just `buf[nread]` (and similarly for `nwrite`). Let's suppose that calls to `piperead` and `pipewrite` happen simultaneously on two different CPUs.

RUNNABLE+code
wait+code
ZOMBIE+code
strud proc+code

`Pipewrite` (6530) begins by acquiring the pipe lock, which protects the counts, the data, and their associated invariants. `Piperead` (6551) 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 (6544). During this loop, it could happen that the bu"er fills (6536). In this case, `pipewrite` calls `wakeup` to alert any sleeping readers to the fact that there is data waiting in the bu"er and then sleeps on `&p->nwrite` to wait for a reader to take some bytes out of the bu"er. Sleep releases `p->lock` as part of putting `pipewrite` to sleep.

Now that `p->lock` is available, `piperead` manages to acquire it and start running in earnest: it finds that `p->nread != p->nwrite` (6556) (`pipewrite` went to sleep because `p->nwrite == p->nread+PIPE_SIZE` (6536)) so it falls through to the for loop, copies data out of the pipe (6563-6567), and increments `nread` by the number of bytes copied. That many bytes are now available for writing, so `piperead` calls `wakeup` (6568) 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 bu"er filled. It marks that process as RUNNABLE.

The pipe code uses separate sleep channels for reader and writer (`p->nread` and `p->nwrite`); this might make the system more efficient in the unlikely event that there are lots of readers and writers waiting for the same pipe. The pipe code sleeps inside a loop checking the sleep condition; if there are multiple readers or writers, all but the first process to wake up will see the condition is still false and sleep again.

Code: Wait, exit, and kill

Sleep and `wakeup` can be used for many kinds of waiting. An interesting example, seen Chapter 0, is the `wait` system call that a parent process uses to wait for a child to exit. In xv6, when a child exits, it does not die immediately. Instead it switches to the ZOMBIE process state until the parent calls `wait` to learn of the exit. The parent is then responsible for freeing the memory associated with the process and preparing the struct `proc` for reuse. 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. Keep in mind 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 (2689) 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 (2628); 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 mark the parent as RUNNABLE, the bop 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 (2640). Before exit reschedules, it reparents all of the exiting process's children, passing them to the initproc (2630-2637). Finally, exit calls sched to relinquish the CPU.

exit+code
sched+code
p->kstack+code
p->pgdir+code
swtch+code

If the parent process was sleeping in wait, the schedule 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 (2634). It records the child's pid and then clears up the struct proc, freeing the memory associated with the process (2668-2676).

The child process could have done most of the cleanup during exit, but it is important that the parent process be the one to free p->kstack and p->pgdir: when the child runs exit, its stack sits in the memory allocated as p->kstack and it uses its own pagetable. They can only be freed after the child process has finished running for the last time by calling swtch (via sched). This is one reason that the schedule procedure 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 (2875) lets one process request that another be terminated. It would be too complex for kill to directly destroy the victim process, since the victim might be executing on another CPU or sleeping while midway through updating kernel data structures. To address these challenges, kill does very little: it just sets the victim's p->killed and, if it is sleeping, 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 waiting 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 (6537) returns if the killed flag is set; eventually the code will return back to trap, which will again check the flag and exit.

Some xv6sleep 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 (4279) 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 process (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 process in turn. This policy is called round robin. Real operating systems implement more sophisticated policies that, for example, allow processes to have priorities. The idea is that a runnable high-priority process will be preferred by the scheduler over a runnable low-priority thread. These policies can become complex quickly because there are often competing goals: for example, the operating might also want to guarantee fairness and high-throughput. In addition, complex policies may lead to unintended interactions such as priority inversion and convoys. Priority inversion can happen when a low-priority and high-priority process share a lock, which when acquired by the low-priority process can cause the high-priority process to not run. A long convoy can form when many high-priority processes are waiting for a low-priority process that acquires a shared lock; once a convoy has formed they can persist for long period of time. To avoid these kinds of problems additional mechanisms are necessary in sophisticated schedulers.

round robin
priority inversion
convoys
thundering herd

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

Scanning the entire process list in wakeupfor processes with a matching channel 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 wakeupcall that structure a rendezvous point or Rendez Many thread libraries refer to the same structure as a condition variable; in that context, the operations sleep and wakeup are called wait and signal. All of these mechanisms share the same behavior: the sleep condition is protected by some kind of lock dropped atomically during sleep.

The implementation of wakeup wakes up all processes that are waiting on a particular channel and it might be the case that many processes are waiting for that particular channel. The operating system will 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 always possible to increment a semaphore but the semaphore value is not allowed to drop below zero: a decrement of a zero semaphore sleeps until another process increments the semaphore and then those two operations cancel out. The integer value typically corresponds to a real count, such as the number of bytes available in a pipe.

bu"er or the number of zombie children that a process has. Using an explicit count as part of the abstraction avoids the Ômissing wakeupÓ problem: there is an explicit count of the number of wakeups that have occurred. The count also avoids the spurious wakeup and thundering herd problems.

Terminating processes and tearing 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 mechanisms for exception handling such as longjmp. Furthermore, there are other events that can cause a sleeping process to be woken up, even though the events it is waiting for has not happened yet. For example when a process is sleeping another process may send a signal to it. In this case the process will return from the interrupted 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 for input from the console but the user doesn't type any input).

Exercises

1. Sleep has to check lk != &ptable.lock to avoid a deadlock(2817-2820). It could eliminate the special case by replacing

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

with

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

Doing this would break sleep. How?

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

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

sult.

4. Fix the race mentioned above between kill and sleep, so that a kill that occurs after the victim's sleep loop checks p->killed but before it calls sleep results in the victim abandoning the current system call.
5. 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.

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 Chapter 0), and stores its data on an IDE disk for persistence (see Chapter 3). The file system addresses several challenges:

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

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

Overview

The xv6 file system implementation is organized in seven layers, shown in Figure 6-1. 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 buffering 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 name and i-number. The pathname layer provides hierarchical path names like /usr/src/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 application programmers.

The file system must have a plan for where it stores nodes and content blocks on the disk. To do so, xv6 divides the disk into several sections, as shown in Figure 6-2. The file system does not use block 0 (it holds the boot sector). Block 1 is called the

```

!"#$%&'#
)"+$"
'--"-.
/011$#.%2%3$

```

Figure 6-1 Layers of the xv6 file system.

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 inodes, with multiple inodes per block. After those come bitmap blocks tracking which data blocks are in use. Most of the remaining blocks are data blocks; each is either marked free in the bitmap block, or holds content for a file or directory. The blocks at the end of the disk hold the logging layer’s log.

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 have 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 buf to the appropriate block on the disk. A kernel thread must release a buf by calling brelse when it is done with it.

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

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

!"#	\$%&(')*"+\$!)#,.-&	+.#.,,,	2222+.#.,,,	/"0
!	1	3				

Figure 6-2 Structure of the xv6 file system. The headerfs.h (3900) contains constants and data structures describing the exact layout of the file system.

used buffer is the one least likely to be used again soon.

Code: Buffer cache

The buffer cache is a doubly-linked list of buffers. The function binit , called by main (1231), initializes the list with the NBUFBuffers in the static array buf (4350-4359). All other access to the buffer cache refer to the linked list via bcache.head, not the buf array.

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

Bread (4402) calls bget to get a buffer for the given sector (4406). If the buffer needs to be read from disk, bread calls iderw to do that before returning the buffer.

Bget (4366) scans the buffer list for a buffer with the given device and sector numbers (4373-4384). If there is such a buffer, and the buffer is not busy, bget sets the B_BUSY flag and returns (4376-4383). If the buffer is already in use, bget sleeps on the buffer to wait for its release. When sleep returns, bget cannot assume that the buffer is now available. In fact, since sleep releases and acquires buf_table_lock , there is no guarantee that b is still the right buffer: maybe it has been reused for a different disk sector. Bget must start over (4382), hoping that the outcome will be different this time.

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 busy: any such buffer can be used. Bget edits the buffer metadata to record the new device and sector number and marks the buffer busy before returning the buffer (4393-4395). Note that the assignment to flags not only sets the B_BUSY bit but also clears the B_VALID and B_DIRTY bits, thus ensuring that bread will read the block data from disk rather than incorrectly using the buffer's previous contents.

Because the buffer cache is used for synchronization, it is important that there is only ever one buffer for a particular disk sector. The assignments (4391-4393) are only safe because bget's loop determined that no buffer already existed for that sector, and bget has not given up buf_table_lock since then.

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

```
binit+code
main+code
NBUF+code
bcache.head+code
B_VALID+code
B_DIRTY+code
B_BUSY+code
bget+code
iderw+code
bget+code
B_BUSY+code
sleep+code
buf_table_lock+code
bget+code
B_BUSY+code
B_VALID+code
B_DIRTY+code
```

Once bread has returned a buffer to its caller, the caller has exclusive use of the buffer and can read or write the data bytes. If the caller does write to the data, it must call bwrite to write the changed data to disk before releasing the buffer. Bwrite (4414) sets the B_DIRTY flag and calls iderw to write the buffer to disk.

When the caller is done with a buffer, it must call brelse to release it. (The name brelse, a shortening of b-release, is cryptic but worth learning: it originated in Unix and is used in BSD, Linux, and Solaris too.) Brelse (4425) moves the buffer to the front of the linked list (4432-4437), dears the B_BUSY bit, and wakes any processes sleeping on the buffer. 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 (starting at bcache.head and following next pointers) will reduce scan time when there is good locality of reference. The scan to pick a buffer to reuse picks the least recently used buffer by scanning backward (following prev pointers).

```

bread+code
bwrite+code
B_DIRTY+code
iderw+code
brelse+code
B_BUSY+code
log
commit

```

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 inconsistent state. For example, depending on the order of the disk writes, a crash during file deletion may either leave a directory entry pointing to a free inode, or it may leave an allocated but unreferenced inode. The latter is relatively benign, but a directory entry that refers to a free inode is likely to cause serious problems after a reboot.

Xv6 solves the problem of crashes during file system operations with a simple version of logging. An xv6 system call does not directly write the on-disk file system data structures. Instead it places a description of all the disk writes it wishes to make in a log on the disk. Once the system call has logged 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.

Why does xv6 not solve the problem of crashes during file system operations? If the crash occurs before the operation commits, then the log on disk will not be marked as complete; the recovery code will ignore it, and the state of the disk will be as if the operation had not even started. If the crash occurs after the operation commits, then recovery will replay all of the operation's writes, perhaps repeating them if the operation had started to write them to the on-disk data structure. In either case, the log makes operations atomic with respect to crashes: after recovery either all of the

operations. Writes appear on the disk, or none of them appear.

group commit
batch
write+code
unlink+code

Log design

The log resides at a known physical location at the end of the disk. 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. The header block also contains the count of logged blocks. Xv6 writes the header block when a transaction commits, but not before and sets the count to zero after copying the logged blocks to the file system. Thus a crash midway through a transaction will result in a count of zero in the log's header block; a crash after a commit will result in a non-zero count.

Each system call code indicates the start and end of the sequence of writes that must be atomic. For efficiency and to allow a degree of concurrency in the file system code, the logging system can accumulate the writes of multiple system calls into each transaction. Thus a single commit may involve the writes of multiple complete system calls. In order to preserve atomicity, the logging system commits only when no file system system calls are underway.

The idea of committing several transaction together is known as group commit. Group commit allows several transactions to run concurrently and allows the file system to batch several disk operations and issue a single disk operation to the disk driver. This allows the disk to schedule the writing of the blocks cleverly and write at the rate of the disk's bandwidth. Xv6's IDE driver doesn't support 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 distinct blocks than there is space in the log. This is not a problem for most system calls, but two of them can potentially write many blocks: write and unlink. A large file write may write many data blocks and many bitmap blocks as well as an inode block; unlinking a large file might write many bitmap blocks and an inode. Xv6's write system call breaks up large writes into multiple smaller writes that fit in the log, and unlink doesn't cause problems because in practice the xv6 file system uses only one bitmap block. 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.

Code: logging

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

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

```
...  
end_op();
```

begin_op (4628) waits until the logging system is not currently committing, and until there is enough free bg space to hold the writes from this call and all currently executing system calls. log.outstanding counts that number of calls; it increments both reserve space and prevents a commit from occurring during this system call. The code conservatively assumes that each system call might write up to MAXOPBLOCKS distinct blocks.

```
begin_op+code  
log_write+code  
bwrite+code  
absorption  
end_op+code  
install_trans+code  
recover_from_log+code  
initlog+code  
bwrite+code  
writei+code
```

log_write (4722) acts as a proxy for bwrite. It records the block's sector number in memory reserving it a slot in the bg on dsk, and marks the buffer B_DIRTY to prevent the block cache from evicting it. The block must stay in the cache until committed: 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 bg. This optimization is often called absorption. It is common that, for example, the dsk block containing inodes of several files is written several times within a transaction. By absorbing several disk writes into one, the file system can save bg space and can achieve better performance because only one copy of the dsk block must be written to disk.

end_op(4653) 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() (4683) copies each block modified in the transaction from the buffer cache to its slot in the bg on dsk. write_head() (4604) writes the header block to dsk: this is the commit point, and a crash after the write will result in recovery replaying the transaction's writes from the bg. install_trans() installs_trans (4572) reads each block from the bg and write it to the proper place in the file system. Finally end_op writes the bg 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 (4618) is called from initlog (4556), which is called during boot before the user process runs. (2794) It reads the bg header and mimics the actions of end_op if the header indicates that the bg contains a committed transaction.

An example use of the bg occurs in filewrite (5752). 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 bg. 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.

```

balloc+code
bfree+code
readsb+code
inode
strut dinode+code
strut inode+code

```

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 bits corresponding to the boot sector, superblock, inode blocks, and bitmap blocks are always set.

The block allocator provides two functions: `balloc` allocates a new disk block, and `bfree` frees a block. `Balloc` (4804) starts by calling `readsb` to read the superblock from the disk (or buffer cache) into `sb`. `balloc` decides which blocks hold the data block free bitmap by calculating how many blocks are consumed by the boot sector, the superblock, and the inodes (using `BBLOCK`). The loop (4812) 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` (4831) finds the right bitmap block and clears the right bit. Again the exclusive use implied by `bread` and `brelse` avoids the need for explicit locking.

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 `size` and list of data block numbers. Or "`inode`" might refer to an in-memory inode which contains a copy of the on-disk inode as well as extra information needed within the kernel.

All of the on-disk inodes are packed into a contiguous area of disk called the inode blocks. Every inode is the same size, so it is easy given a number `n`, to find the `n`th 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` (3926). The type field distinguishes 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 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` (4012) is the in-memory copy of a `struct dinode` on disk. The kernel stores an inode in memory only if there are 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.

A 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 references 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 lock it) and reads the inode from the disk, if it has not already been read. `iunlock` releases the lock on the inode. Separating acquisition of inode pointers from locking helps avoid deadlock in some situations, for example during directory lookup. Multiple processes can hold a C pointer to an inode returned by `iget`, but only one process can lock the inode at a time.

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

Code: Inodes

To allocate a new inode (for example when creating a file), xv6 calls `ialloc` (4953). `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` (4970). The correct operation of `ialloc` depends on the fact that only one process at a time can be holding a reference to it: `ialloc` can be sure that some other process does not simultaneously see that the inode is available and try to claim it.

`iget` (5004) 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. (5013-5017) As `iget` scans, it records the position of the first empty slot (5018-5019) 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` (5053) uses a now-familiar sleep loop to wait for `ip->flag`'s `I_BUSY` bit to be clear and then sets it (5062-5064). Once `ilock` has exclusive access to the inode, it can read the inode metadata from the disk (more likely, the buffer cache) if needed. The function `iunlock` (5085) clears the `I_BUSY` bit and wakes any processes sleeping in `ilock`.

`iput` (5108) releases a C pointer to an inode by decrementing the reference count

```

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

```

(5124) If this is the last reference the `inode` slot in the `inode cache` is now free and can be re-used for a different inode.

If `iput` sees that there are no C pointer references to an inode and that the inode has no links to it (occurs in no directory), then the inode and its data blocks must be freed. `iput` relocks the inode; calls `itrunc` to truncate the file to zero bytes, freeing the data blocks; sets the inode type to 0 (`unallocated`); writes the change to disk; and finally unlocks the inode (5111-5123).

The locking protocol in `iput` in the case in which it frees the inode deserves a closer look. First, when locking `ip` by setting `I_BUSY`, `iput` assumes that it is unlocked. This must be the case: the caller is required to unlock `ip` before calling `iput`, and no other process can lock this inode because no other process can get a pointer to it. That is because in this code path, the inode has no reference, no links (i.e., no pathname refers to it), and is not (yet) marked free. The second part worth examining is that `iput` temporarily releases (5116) and reacquires (5120) the `inode cache` lock, because `itrunc` and `iupdate` will sleep during disk I/O. But we must consider what might happen while the lock is not held. Specifically, once `iupdate` finishes, the on-disk inode is marked as free and a concurrent call to `ialloc` might bind it and reallocate it before `iput` can finish. `ialloc` will return a reference to the block by calling `iget`, which will bind `ip` in the cache see that its `I_BUSY` flag is set, and sleep. Now the in-core inode is out of sync compared to the disk: `ialloc` reinitialized the disk version but relies on the caller to load it into memory during `ilock`. In order to make sure that this happens, `iput` must clear not only `I_BUSY` but also `I_VALID` before releasing the inode lock. It does this by zeroing flags (5121).

`iput()` can write the disk. This means that any system call that uses the file system may write the disk, even calls like `read()` that appear to be read-only. This, in turn, means that even read-only system calls must be wrapped in transactions if they use the file system.

Code: Inode content

The on-disk inode structure, `struct dinode`, contains a size and an array of block numbers (see Figure 6-4). The inode data is found in the blocks listed in the `dinode->addrs` array. The first `NDIRECT` blocks of data are listed in the `firstNDIRECT` 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 ($NDIRECT \times BSIZE$) bytes of a file can be read from blocks listed in the inode, while the next 64kB ($NINDIRECT \times BSIZE$) bytes can only be read after consulting the indirect block. This is a good on-disk representation but a complex one for clients. The function `bmap` manages the representation so that higher-level routines such as `readi` and `writei`, which we will see shortly, `bmap` returns the disk block number of the first data block for the inode `ip`. If `ip` does not have such a block yet, `bmap` allocates one.

The function `bmap` begins by picking the easy case: the first `NDIRECT` blocks are listed in the inode itself (5165-5169). The next `NINDIRECT` blocks are listed in

```
iput+code
itrunc+code
iput+code
iput+code
itrunc+code
iupdate+code
ialloc+code
iget+code
I_BUSY+code
I_VALID+code
strud dinode+code
NDIRECT+code
dired blocks
NINDIRECT+code
indirect block
BSIZE+code
bmap+code
readi+code
writei+code
bmap+code
NDIRECT+code
NINDIRECT+code
```

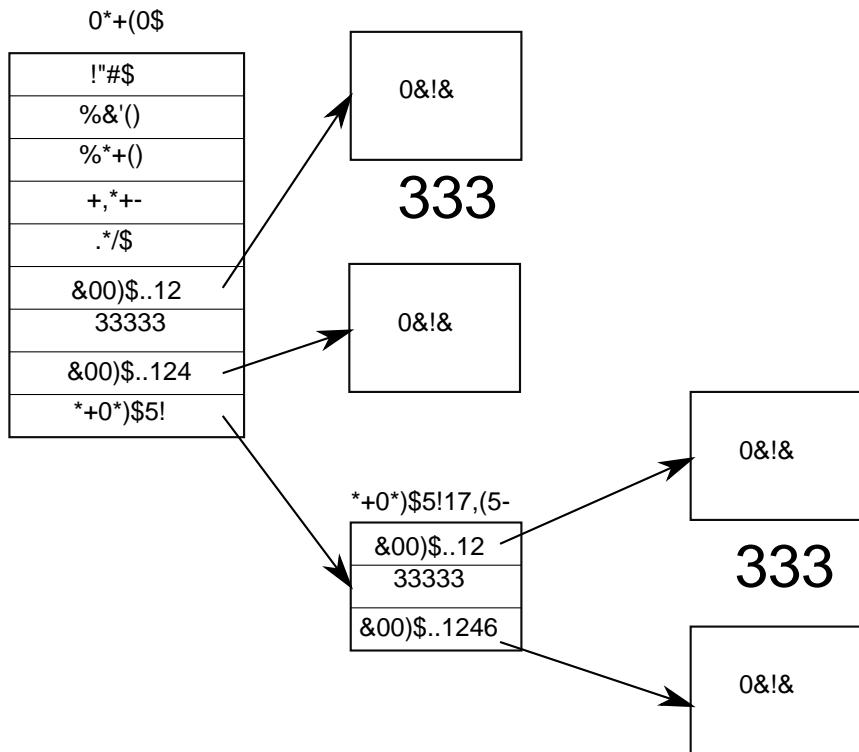


Figure 6-4 The representation of a file on disk.

the indirec block at ip->addrs[NDIRECT]. Bmap reads the indirec block (5176) and then reads a block number from the right position within the block (5177). If the block number exceeds NDIRECT+NINDIREC⁶ happens; writei contains the check that prevents this from happening(5315)

Bmap allocates blocks as needed. An ip->addrs[] or indirec entry of zero indicates that no block is allocated. As bmapencounters zeros, it replaces them with the numbers of fresh blocks, allocated on demand. (5166-5167, 5174-5175)

itrunc frees a file's blocks, resetting the node's size to zero. Itrunc (5206) starts by freeing the direct blocks (5212-5217), then the ones listed in the indirec block (5222-5225) and finally the indirec block itself (5227-5228).

Bmap makes it easy for readi and writei to get at an inode's data. Readi (5252) 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 (5263-5264) while reads that start at or cross the end of the file return fewer bytes than requested(5265-5266). The main loop processes each block of the file copying data from the buffer into dst (5268-5273). writei (5302) 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 (5315-5316) the loop copies data into the buffers instead of out (5321) and if the write has extended the file, writei must update its size (5326-5329).

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 `stat` (4773) copies inode metadata into the `stat` structure which is exposed to user programs via the `stat` system call.

```
stat+code
stat+code
T_DIR+code
strut dirent+code
DIRSIZ+code
dirlookup+code
iget+code
.+code
..+code
dirlink+code
dirlookup+code
nameiparent+code
namex+code
skipel+code
ilock+code
nameiparent+code
namex+code
```

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` (3950) which contains a name and an inode number. The name is at most `DIRSIZ(14)` characters; if shorter, it is terminated by a NUL (0) byte. Directory entries with inode number zero are free.

The function `dirlookup` (5361) searches a directory for an entry with the given name. If it finds one, it returns a pointer to the corresponding inode unlocked and sets `*poff` to the byte offset of the entry within the directory, in case the caller wishes to edit it. If `dirlookup` finds an entry with the right name, it updates `*poff`, releases the block, and returns an unlocked inode obtained via `iget`. `Dirlookup` is the reason that `iget` returns unlocked inodes. The caller has locked `dp`, so if the backup was for `..`, an alias for the current directory, attempting to lock the node before returning would try to re-lock `dp` and deadlock. (There are more complicated deadlock scenarios involving multiple processes and `..`, an alias for the parent directory; `..` is not the only problem.) The caller can unlock `dp` and then lock `ip`, ensuring that it only holds one lock at a time.

The function `dirlink` (5402) 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 (5408-5412). The main loop reads directory entries looking for an unallocated entry. When it finds one, it stops the loop early (5372-5373) 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 `off_set` (5422-5425).

Code: Path names

Path name lookup involves a succession of calls to `dirlookup`, one for each path component. `Namei` (5540) 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 generalized function `namex` to do the real work.

`Namex` (5505) starts by deciding where the path evaluation begins. If the path begins with a slash, evaluation begins at the root; otherwise the current directory (5509-5512). Then it uses `skipel` to consider each element of the path in turn (5514). 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 (5515-5519). (Locking `ip` is necessary not because `ip->type` can change underfoot 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` (5520-5524). Finally, the loop looks for

the path element using dirlookup and prepares for the next iteration by setting ip = next (5525-5530). When the loop runs out of path elements, it returns ip.

```
dirlookup+code
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
sys_link+code
sys_unlink+code
```

File descriptor layer

One of the cool aspects 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 (4000) which is a wrapper around either an inode or a pipe, plus an i/o set. 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 sets. On the other hand, a single open file (the same struct file) can appear multiple times in one process's ftable 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 (5625) scans the file table for an unreferenced file (f->ref == 0) and returns a new reference. fileread (5652) increments the reference count; and fileclose (5664) decrements it. When a file's reference count reaches zero, fileclose releases the underlying pipe or node according to the type.

The functions filestat, fileread, and filewrite implement the stat, read, and write operations on files. Filestat (5702) is only allowed on nodes 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 node implementation. If the file represents an inode, fileread and filewrite use the i/o set as the i/o set for the operation and then advance it (5725-5726, 5765-5766). Pipes have no concept of i/o sets. Recall that the inode functions require the caller to handle locking (5705-5707, 5724-5727, 5764-5778). The inode locking has the convenient side effect that the read and write operations are updated atomically so that multiple writing to the same file simultaneously cannot overwrite each other's data, though their writes may end up interlaced.

Code: System calls

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

The functions sys_link and sys_unlink edit directories, creating or removing reference to nodes. They are another good example of the power of using transac-

tions. `Sys_link` (5913) begins by fetching its arguments, two strings `old` and `new` (5918). Assuming `old` exists and is not a directory (5922-5925) `sys_link` increments its `ip->nlink` count. Then `sys_link` calls `nameiparent` to find the parent directory and final path element of `new` (5938) and creates a new directory entry pointing at `old`'s inode (5941). The new parent directory must exist and be on the same device as the existing inode: inode numbers only have a unique meaning on a single disk. If an error like this occurs, `sys_link` must go back and decrement `ip->nlink`.

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

`Sys_link` creates a new name for an existing inode. The function `create` (6057) creates a new name for a new inode. It is a generalization of the three file creation system calls: `open` with the `O_CREAT` flag makes a new ordinary file, `mkdir` makes a new directory, and `mkdev` makes a new device file. Like `sys_link`, `create` starts by calling `nameiparent` to get the inode of the parent directory. It then calls `dirlookup` to check whether the name already exists (6067). If the name does exist, `create`'s behavior depends on which system call it is being used for: `open` has different semantics from `mkdir` and `mkdev`. If `create` is being used on behalf of `open` (`type == T_FILE`) and the name that exists is itself a regular file, then `open` treats that as a success, so `create` does too (6071). Otherwise it is an error (6072-6073). If the name does not already exist, `create` now allocates a new inode with `alloc` (6076). 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 (6089). `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` (6101) 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_CREAT` flag, it calls `create` (6114). Otherwise, it calls `namei` (6120). `Create` returns a locked inode, but `namei` does not, so `sys_open` must lock the inode itself. This provides a convenient place to check that directories are only opened for reading not writing. Assuming the inode was obtained one way or the other, `sys_open` allocates a file and a file descriptor (6132) and then puts it in the file (6142-6146). 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 system. The function `sys_pipe` connects that implementation to the file system by providing a way to create a pipe pair. Its argument is a pointer to space for two integers, where it will record the two new file descriptors. Then it allocates the pipe and installs the file descriptors.

```

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
alloc+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
sys_pipe+code

```

Real world

The buffer cache in a real-world operating system is significantly more complex than xv6, but it serves the same two purposes: caching and synchronizing access to the disk. Xv6's buffer cache like V6 uses a simple least recently used (LRU) eviction policy; there are many more complex policies that can be implemented and good for some workloads and not as good for others. A more efficient LRU cache would eliminate the linked list, instead using a hash table for lookups and a map for LRU evictions. 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 node free lists, looking for and resolving inconsistencies. Scavenging can take hours for large file systems, and there are situations where it is not possible to 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 during each lookup. This is reasonable when directories are only a few disk blocks, but is expensive for directories holding many files. Microsoft Windows' NTFS, Mac OS X's HFS, and Solaris' ZFS just to name a few implement a directory as an n-disk balanced 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 system has special hardware that uses redundancy to mask disk failures, perhaps the operating system sees failures so infrequently that panicking is okay. On the other hand, operating systems using plain disks should expect failures and handle them more gracefully so that the loss of a block in one file doesn't affect the use of the rest of the file system.

Xv6 requires that the file system be on one disk device and not change in size. As large database and multimedia files drive storage requirements ever higher, operating systems are developing ways to eliminate the one disk per file system bottleneck. The basic approach is to combine many disks into a single logical disk. Hardware solutions such as RAID are still the most popular, but the current trend is moving toward implementing as much of this logic in software as possible. These software implementations typically 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 able system that can do the same: the fixed-size array of inode blocks used by xv6 would not work well in such environments. Separating disk

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

fileread+code
filewrite+code

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 resource to be accessed with the same system calls as disk storage: named pipes, network connections, remotely-accessed network file systems, and monitoring and control interfaces such as /proc. Instead 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 user-level file systems provide functions that turn those calls into network RPCs and wait for the response before returning.

Exercises

1. Why panic in malloc? Can xv6 recover?
2. Why panic in free? Can xv6 recover?
3. Why doesn't filepanic 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 (calls to iunlock(ip) and dirlink). Will the link be created correctly? Why or why not?
5. create makes four function calls (one to malloc 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 input(cp->cwd), which might try to lock cp->cwd yet postponing iunlock(ip) until after the input would not cause deadlocks. Why not?

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 (eg., 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 (eg., 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.

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

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

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

Processor and memory

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

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

instruction pointer
control registers
segment registers

written quickly, in a single CPU cycle.

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

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

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

Intel Core 2 Duo E7200 at 2.53 GHz

TODO: Plug in non-made-up numbers!

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

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

I/O

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

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

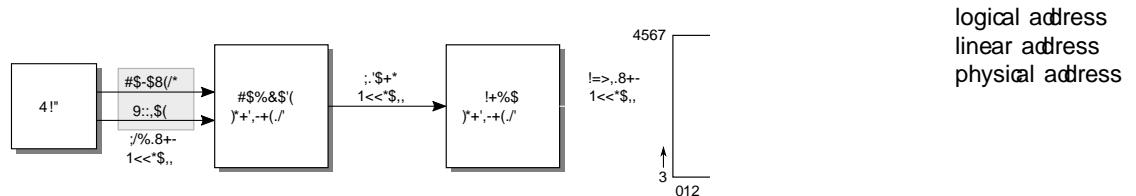


Figure B-1 The relationship between logical, linear, and physical addresses.

Appendix B

boot loader
boot loader
real mode

The boot loader

When an x86 PC boots, it starts executing a program called the BIOS, which is stored in non-volatile memory on the motherboard. The BIOS's job is to prepare the hardware and then transfer control to the operating system. Specifically it transfers control to code loaded from the boot sector, the first 512-byte sector of the boot disk. The boot sector contains the boot loader instructions that load the kernel into memory. The BIOS loads the boot sector at memory address 0x7c00 and then jumps (sets the processor's CS register) to that address. When the boot loader begins executing the processor is simulating an Intel 8088, and the boot loader's job is to put the processor in a more modern operating mode to load the xv6 kernel from disk into memory and then to transfer control to the kernel. The xv6 boot loader comprises two source files, one written in a combination of 16-bit and 32-bit x86 assembly (bootasm.S (8900)) and one written in C (bootmain.c; (9000)).

Code: Assembly bootstrap

The `jmp` instruction in the boot loader is `cli` (8912) which disables processor interrupts. Interrupts are a way of hardware devices to invoke operating system functions called interrupt handlers. The BIOS is a tiny operating system, and it might have set up its own interrupt handlers as part of 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 address to memory. The segment registers %cs %ds %es and %ss provide the additional bits necessary to generate 20-bit memory addresses from 16-bit registers. When a program refers to a memory address, the processor automatically adds 16 times the value of one of the segment registers; these registers are 16 bits wide. Which segment register is usually implicit in the kind of memory reference: instruction fetches use %cs, data reads and writes use %ds and %ss, stack reads and writes use %ss.

boot loader
logical address
linear address
virtual address
protected mode
segment descriptor table
gdt+code

Xv6 pretends that an x86 instruction uses a virtual address for its memory operands, but an x86 instruction actually uses a logical address (see Figure B-1). A logical address consists of a segment selector and an offset and is sometimes written as segment:offset. More often, the segment is implicit and the program only directly manipulates the offset. The segmentation hardware performs the translation described above to generate a linear address. If the paging hardware is enabled (see Chapter 2), it translates linear addresses to physical addresses; otherwise the processor uses linear addresses as physical addresses.

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

The BIOS does not guarantee anything about the contents of %ds %es %ss so first order of business after disabling interrupts is to set %ax to zero and then copy that zero into %ds %es and %ss (8915-8918).

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: $0xfffff0 + 0xffff = 0x10ffef$, but virtual address 0xfffff : 0xffff on the 8088 referred to physical address 0x0ffef. Some early software relied on the hardware ignoring the 21st address bit, so when Intel introduced processors with more than 20 bits of physical address, IBM provided a compatibility hack that is a requirement for PC-compatible hardware. If the second bit of the keyboard controller's output port is low, the 21st physical address bit is always cleared; if high, the 21st bit acts normally. The boot loader must enable the 21st address bit using I/O to the keyboard controller on ports 0x64 and 0x60 (8920-8936).

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 processor since the 80286 have a protected mode, which allows physical addresses to have many more bits, and (since the 80386) a 32-bit mode that cause registers, virtual addresses, and most integer arithmetic to be carried out with 32 bits rather than 16. The xv6 boot sequence enables protected mode and 32-bit mode as follows.

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

xv6 makes almost no use of segments; it uses the paging hardware instead as Chapter 2 describes. The boot loader sets up the segment descriptor table gdt (8982-8985) so that all segments have a base address of zero and the maximum possible limit (four gigabytes). The table has a null entry, one entry for executable code and one en-

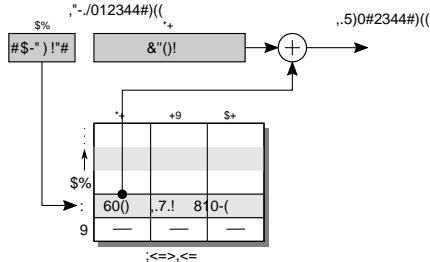


Figure B-2 Segments in protected mode.

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

The boot loader executes an `lgdt` instruction (8941) to load the process's global descriptor table (GDT) register with the value `gdtdesc` (8987-8989), 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_P) in register `%cr0` (8942-8944). 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 (8953) which allows a code segment selector to be specified. The jump continues execution at the next line (8956) but in doing so sets `%cst` 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 passed the processor through an evolution from 8088 through 80286 to 80386.

The boot loader's transition in 32-bit mode is to initialize the data segment registers with `SEG_KDATA` (8958-8961). Logical addresses now map directly to physical addresses. The only step left before executing C code is to set up a stack in an unused region of memory. The memory from 0xa0000 to 0x100000 is typically littered with device memory regions, and the xv6 kernel expects to be placed at 0x100000. The boot loader itself is at 0x7c00 through 0x7d00. Essentially any other section of memory would be a fine location for the stack. The boot loader chooses 0x7c00 (known in this file as `$start`) as the top of the stack; the stack will grow down from there toward 0x0000 away from the boot loader.

Finally the boot loader calls the C function `bootmain` (8968). `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 (8970-8976). 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 (8977-8978). 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

```
readseg+code  
stosb+code  
_start+code  
entry+code
```

The C part of the boot loader, bootmain.c (9000), expects to find a copy of the kernel executable on the disk starting at the second sector. The kernel is an ELF format binary, as we have seen in Chapter 2. To get access to the ELF headers, bootmain loads the first 4096 bytes of the ELF binary (9014). It places the in-memory copy at address 0x10000.

The next step is a quick check that this probably is an ELF binary, and not an uninitialized disk. Bootmain reads the section 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 (9038) and calls stosb to zero the remainder of the segment (9040). 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 destination addresses that look like 0x801xxxxx; you can see examples in kernel.asm. This address is configured in kernel.ld. 0x80100000 is a relatively high address, towards the end of the 32-bit address space; Chapter 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 assumes that there is physical memory at this lower address. At this point in the boot process, however, paging is not enabled. Instead, kernel.ld specifies that the ELF paddr starts 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 last step is to call the kernel entry point, which is the instruction at which the kernel expects to start executing. For xv6, the entry address is 0x10000c:

```
# objdump -f kernel

kernel:      file    format elf32-i386
architecture: i386, flags 0x00000012:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

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

Real world

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

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

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

Exercises

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

Index

. , 81, 83
. , 81, 83
/init , 27, 35
_binary_initcode_size , 25
_binary_initcode_start , 25
_start , 96
absorption, 76
acquire , 51Ð52, 54
addl , 26
address space 21

KERNBASE
 kernel 7, 19
 kernel mode 19, 38
 kernel space 7, 19
 kfree , 33
 kinit1 , 33
 kinit2 , 33
 kmap 32
 kvmalloc , 30, 32
 lapicinit , 44
 linear address, 93Ð94
 links, 15
 loaduvm 35
 lock, 49
 log, 74
 log_write , 76
 logical address, 93Ð94
 main, 23, 26, 32Ð33, 40, 45, 73
 malloc , 10
 mappages, 32
 memory-mapped I/O, 91
 microkernel 20
 mkdkey 83
 mkdir, 83
 monolithic kernel 17, 19
 mpmain26
 multiplex, 57
 namej 25, 34, 83
 nameiparent, 81, 83
 namesx81
 NBU_F73
 NDIRECT79
 NINDIRECT79
 O_CREAT83
 open, 82Ð83
 outb, 44
 p->context , 24, 26, 60
 p->cwd 25
 p->kstack , 22, 67
 p->name25
 p->pgdir , 22, 67
 p->state , 22
 p->sz, 43
 p->xxx, 21
 page 29
 page directory, 29
 page table entries (PTEs), 29
 page table pages, 29
 panic, 42
 parent process, 8
 path, 14
 persistence71
PGROUNDDB
 physical address, 21, 93
PHYSTOB2D33
 picenable , 45
 pid , 8, 24
 pipe, 13
 piperead, 65
 pipewrite , 65
 polling, 46
 popal, 26
 popcli , 54
 popl, 26
 printf , 9
 priority inversion, 68
 privileged instructions, 19
 proc->killed , 42
 process, 7Ð8, 20
 program counter, 89
 programmable interrupt
 controller (PIC), 44
 protected mode, 94Ð95
 ptable , 54
 ptable.lock , 59Ð60, 64Ð65
PTE_P29
PTE_U26, 30Ð32
PTE_W29
 pushcli , 54
 race condition, 50
 read, 82
 ready , 35, 79Ð80
 readsb, 77
 readseg, 96
 real mode, 93
 recover_from_log , 76
 recursive bcks, 52
 release , 52, 54Ð55
 ret , 26
 root, 14
 round robin, 68
RUNNABLE, 60, 64Ð66
 sbrk, 10, 34
 sched, 58Ð60, 64, 67
 scheduler , 26, 59Ð60
 sector 45
SEG_KCP41
SEG_KDAT95
SEG_TS96
SEG_UCODE
SEG_UDATA
 seginit , 36
 segment descriptor table 94
 segment registers, 90
 sequence coordination, 61
 setupkvm 25Ð26, 32, 35
 signal , 69
 skipelem, 81
 sleep , 54, 59, 62Ð64, 73
SLEEPING64Ð65
 stat , 81Ð82
 stati , 81Ð82
 sti , 44, 54
 stosb , 96
 struct buf , 45
 struct context , 58
 struct dinode , 77, 79
 struct dirent , 81
 struct elfhdr , 34
 struct file , 82
 struct inode , 77
 struct pipe , 65
 struct proc , 21, 66
 struct run , 33
 struct spinlock , 51
 struct trapframe , 25
 superblock, 72
 switchuvm, 26, 40, 44, 60
 swtch, 26, 58Ð60, 67
 sys_exec, 40
SYS_exec27, 42
 sys_link , 82Ð83
 sys_mkdir, 83
 sys_mknod83
 sys_open 83
 sys_pipe, 83
 sys_sleep, 54
 sys_unlink , 82
 syscall , 42
 system call, 7
T_DEV80
T_DIR81
T_FILE83
T_SYSCALL27, 40, 42
 tf->trapno , 42
 thread 21
 thundering herd, 68
 ticks , 54
 tickslock , 54
 time-share 8, 17

timer.c , 44
transaction, 71
trap , 41Ð42, 45Ð46, 58
trapret , 24Ð26, 42
traps, 38
tvinit, 40
type cast 33
unlink , 75
use memory 21
use mode 19, 38
use space 7, 19
userinit , 25Ð26
ustack, 35
V2P_WQ3
vectors[i] , 40
virtual address, 21, 94
wait channel 62
wait , 9, 60, 66
wakeup 45, 54, 62, 64Ð65
wakeup1 65
walkpgdir , 32, 35
write , 75, 82
writei , 76, 79Ð80
xchg, 52, 54
yield , 58Ð60
ZOMBIE 66

