Chapter 0

Operating system interfaces

Computers are simple machines of enormous complexity. On the one hand, a processor can do very little: it just executes a single instruction from memory and repeats, billions of times per second. On the other hand, the details of how it does this and how software is expected to interact with the hardware vary wildly. The job of an operating system is to address both of these problems. An operating system creates the illusion of a simple machine that does quite a bit for the programs that it runs. It manages the low-level hardware, so that, for example, a word processor need not concern itself with which video card is being used. It also multiplexes the hardware, allowing many programs to share the computer and run (or appear to run) at the same time. Finally, operating systems provide controlled ways for programs to interact with each other, so that programs can share data or work together.

This description of an operating system does not say exactly what interface the operating system provides to user programs. Operating systems researchers have experimented and continue to experiment with a variety of interfaces. Designing a good interface turns out to be a difficult challenge. 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, application writers want to offer many features to users. The trick in resolving this tension is to design interfaces that rely on a few mechanism that can be combined in ways 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. The Unix operating system provides an an example of 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.

Xv6 takes the 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 correspond to the machine instructions that implement the program's computation. The data corresponds to the data structures that the program uses to implement its computation. The stack allows the program to invoke procedure calls and run the computation.

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

The kernel uses the CPU's hardware protection mechanisms to ensure that each process executing in user space can access only its own memory. The kernel executes with the hardware privileges required to implement these protections; user programs execute without those privileges. When a user program invokes a system call, the hardware raises the privilege level and starts executing a pre-arranged function in the kernel. Chapter 3 examines this sequence in more detail.

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

System call	**Description**
fork()	Create process
exit()	Exit process
wait()	Wait for a child
kill(pid)	Send a signal to process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(*argv)	Load program
sbrk(n)	Grow process's memory with n bytes
open(s, flags)	Open a file with mode specified in flags
read(fd, buf, n)	Read n byes from an open file into buf
write(fd, buf, n)	Write n bytes from an open file into fd
close(fd)	Release fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(s)	Change directory to directory s
mkdir(s)	Create a new directory s
mknod(s, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(s1, s2)	Create another name (s2) for the file s1
unlink(s)	Remove a name

The rest of this chapter outlines xv6's services—processes, memory, file descriptors, pipes, and a file system—by using the system call interface in small code examples, and explaining how the shell uses the system call interface. The shell's use of the system calls illustrates how carefully the system calls have been designed.

The shell is an ordinary program that reads commands from the user and executes them. It is the main interactive way that users use traditional Unix-like systems. The fact that the shell is a user program, not part of the kernel, means that it is easily replaced. In fact, modern Unix systems have a variety of shells to choose from, each with its own syntax and semantics. The xv6 shell is a simple implementation of the essence of the Unix Bourne shell. It's implementation can be found at sheet (6850).

Code: Processes and memory

An xv6 process consists of user-space memory (instructions, data, and stack) and a kernel process data structure. Xv6 provides time-sharing: it transparently switches the available CPUs among the set of processes waiting to execute. When a process is not executing, xv6 saves its CPU registers, restoring them when it next runs the process. Each process can be uniquely identified by a positive integer called its process identifier, or *pid*.

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

```
int pid;

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

The `exit` system call causes the calling process to exit (stop executing). The `wait` system call waits for one of the calling process's children to exit and returns the pid of the child that exited. In the example, the output lines

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

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

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

Note that the parent and child were executing with different memory and different registers: changing a variable in the parent does not affect the same variable in the child, nor does the child affect the parent. The main form of direct communication between parent and child is `wait` and `exit`.

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 are instructions, which part is data, at which instruction to start, etc.. The format xv6 uses is called the ELF format, which Chapter 1 discusses in more detail. When `exec` succeeds, it does not return to the calling program; instead, the instructions loaded from the file start executing at the entry point declared in the ELF header. `Exec` takes two arguments: the name of the file containing the executable and an array of string arguments. For example:

```
char *argv[3];

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

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

The xv6 shell uses the above calls to run programs on behalf of users. The main structure of the shell is simple; see `main` on line (7001). The main loop reads the input on the command line using `getcmd`. Then it calls `fork`, which creates another running shell program. The parent shell calls `wait`, while the child process runs the command. For example, if the user had typed "echo hello" at the prompt, `runcmd` would have been called with "echo hello" as the argument. `runcmd` (6906) runs the actual command. For the simple example, it would call `exec` on line (6926), which loads and starts the program `echo`, changing the program counter to the first instruction of `echo`. If `exec` succeeds then the child will be running `echo` and the child will not execute the next line of `runcmd`. Instead, it will be running instructions of `echo` and at some point in the future, `echo` will call `exit`, which will cause the parent to return from `wait` in `main` (7001). You might wonder why `fork` and `exec` are not combined in a single call; as we will see later, the choice of having separate calls for creating a process and loading a program is clever.

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

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

Code: File descriptors

A file descriptor is a small integer representing a kernel-managed object that a process may read from or write to. A file descriptor is obtained by calling `open` with an pathname as argument. The object by the pathname may be a data file, a directory, a pipe, or the console. It is conventional to call whatever object a file descriptor refers to a file. 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 (7007), 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 open file corresponding to the file descriptor `fd`, copies them into `buf`, and returns the number of bytes copied. Every file descriptor has an offset associated with it. `Read` reads data from the current file offset and then advances that offset by the number of bytes read: a subsequent `read` will return the bytes following the ones returned by the first `read`. When there are no more bytes to read, `read` returns zero to signal the end of the file.

The call `write(fd, buf, n)` writes `n` bytes from `buf` to the open file named by 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 `echo`) copies data from its standard input to its standard output. If an error occurs, it writes a message on 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 `echo` doesn't know whether it is reading from a file, console, or whatever. Similarly `echo` 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 `echo`.

The `close` system call releases a file descriptor, making it free for reuse by a future `open`, `pipe`, or `dup` system call (see below). An important rule in Unix is that the kernel must always allocates the lowest-numbered file descriptor that is unused by the calling process.

This rule and how `fork` works makes I/O redirection work well. `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. `Exec` replaces the calling process's memory but preserves its file table. This behavior allows the shell to implement I/O redirection by forking, reopening chosen file descriptors, and then execing the new program. Here is a simplified version of the code a shell runs for the command `cat <input.txt`:

```
char *argv[2];

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

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

The code for I/O redirection in the xv6 shell works exactly in this way; see the case at (6930). Recall that at this point in the code the shell already forked the child shell and that `runcmd` will call `exec` to load the new program. Now it should be clear why it is a good idea that `fork` and `exec` are separate calls. This separation allows the shell to fix up the child process before the child runs the intended program.

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

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

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

The `dup` system call duplicates an existing file descriptor onto a new one. Both file descriptors share an offset, just as the file descriptors duplicated by `fork` do. This is another way to write `hello world` into a file:

```
close(2);
dup(1);  // uses 2, assuming 0 and 1 not available
write(1, "hello ", 6);
write(2, "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 the following one correctly (`2>` means redirect file descriptor 2): `ls existing-file non-existing-file > tmp1 2> tmp1`. Both the name of the existing file and the error message for the non-existing file will show up in the file `tmp1`. The xv6 shell doesn't support I/O redirection for the error file descriptor, but now you can implement it.

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

Code: Pipes

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

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

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

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

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

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

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

The xv6 shell implements pipes in similar manner as the above code fragment; see (6950). The child process creates a pipe to connect the left end of the pipe with the right end of the pipe. Then it calls `runcmd` for the left part of the pipe and `runcmd` for the right end of the pipe, and waits for the left and the right end to finish, by calling `wait` twice. The right end of the pipe may be a command that itself includes a pipe (e.g., `a | b | c`), which itself forks two new child processes (one for `b` and one for `c`).

Thus, the shell may create a tree of processes. The leaves of this tree are commands and the interior nodes are processes that wait until the left and right children complete. In principle, you could have the interior nodes run the left end of a pipe, but doing so correctly will complicate the implementation.

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

```
echo hello world | wc
```

could also be implemented without pipes as

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

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

Code: File system

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

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

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

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

The `open` system call evaluates the path name of an existing file or directory and prepares that file for use by the calling process.

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

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

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

The fstat system call queries an open file descriptor to find out what kind of file it is. It fills in a struct stat, defined in stat.h as:

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

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

In xv6, a file's name is separated from its content; the same content, 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, but not necessarily the underlying inode. Adding

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

to the last code sequence will not remove the inode, because it is still accessible as b. In order to remove or reuse an inode, xv6 requires not only that all its names have been unlinked but also that there are no file descriptors referring to it. Thus,

```
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.

The xv6 shell doesn't directly support any calls for manipulating the file system. User commands for file system operations are implemented as separate user-level programs such as mkdir, ln, rm, etc. This design allows anyone to extend the shell with new user commands. In hint-sight this plan seems the obvious right one, but when Unix was designed it was common that such commands were built into the shell.

The one exception is cd, which is a build in command; see line (7016). The reason is that cd must change the current working directory of the shell itself. If cd were run as a regular command, then the shell would fork a child process, the child process would run `cd`, change the *child*'s working directory, and then return to the parent. The parent's (i.e., the shell's) working directory would not change.

Real world

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

Xv6, like Unix before it, has a very simple interface. It doesn't implement modern features like networking or computer graphics. The various Unix derivatives have many more system calls, especially in those newer areas. Unix's early devices, such as terminals, are modeled as special files, like the `console` device file discussed above. The authors of Unix went on to build Plan 9, which applied the "resources are files" concept to even these modern facilities, representing networks, graphics, and other resources as files or file trees.

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

This book examines how xv6 implements its Unix-like interface, but the ideas and concepts apply to more than just Unix. Any operating system must multiplex processes onto the underlying hardware, isolate processes from each other, and provide mechanisms for controlled inter-process communication. After studying this book, 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

Bootstrap

Hardware

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

The execution engine is useless without the ability to store and modify program data. The simplest, fastest storage for data is provided by the processor's register set. A register is a storage cell inside the processor itself, capable of holding a machine word-sized value (typically 16, 32, or 64 bits). Data stored in registers can typically be read or written quickly, in a single CPU cycle. The 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, segment selector, and descriptor table registers are important to any operating system, as we will see in this chapter. The floating-point and debug registers are less interesting and not used by xv6.

Registers are very fast but very expensive in bulk. 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. A typical x86 processor has at most a kilobyte of registers, but a typical PC today has gigabytes of main memory. Because of the enormous differences in both access speed and size between registers and main memory, most processors, including the x86, store copies of recently-accessed sections of main memory in on-chip cache memory. The cache memory serves as a middle ground between registers and memory both in access time and in size. Today's x86 processors typically have two levels of cache, a small first-level cache with access times relatively close to the processor's clock rate and a larger second-level cache with access times in between the first-level cache and main memory. This table shows actual numbers for an Intel Core 2 Duo system:

Intel Core 2 Duo E7200 at 2.53 GHz

TODO: Plug in non-made-up numbers!

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

For the most part, x86 processors hide the cache from the operating system, so we can think of the processor as having just two kinds of storage—registers and memory—and not worry about the distinctions between the different levels of the memory hierarchy. The exceptions—the only reasons an x86 operating system needs to worry about the memory cache—are concurrency (Chapter 4) and device drivers (Chapter 6).

One reason memory access is so much slower than register access is that the memory is a set of chips physically separate from the processor chip. To allow the processor to communicate with the memory, there is a collection of wires, called a bus, running between the two. A simple mental model is that some of the wires, called lines, carry address bits; some carry data bits. To read a value from main memory, the processor 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 processor sends appropriate bits on the address and data lines and a 1 on the "write" line for a prescribed amount of time. This model is an accurate description of the earliest x86 chips, but it is a drastic oversimplification of a modern system. Even so, thanks to the processor-centric view the operating system has of the rest of the computer, this simple model suffices to understand a modern operating system. The details of modern I/O buses are the province of computer architecture textbooks.

Processors must communicate not just with memory but with hardware devices too. The x86 processor provides special `in` and `out` instructions that read and write values from device addresses called 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 a device port and 1 meant read/write from main memory.

Many computer architectures have no separate device access instructions. Instead the devices have fixed memory addresses and the processor communicates with the device (at the operating system's behest) by reading and writing values at those addresses. In fact, modern x86 architectures use this technique, called memory-mapped

I/O, for most high-speed devices such as network, disk, and graphics controllers. For reasons of backwards compatibility, though, the old `in` and `out` instructions linger, as do legacy hardware devices that use them, such as the IDE disk controller, which we will see shortly.

Bootstrap

When an x86 PC boots, it starts executing a program called the BIOS, which is stored in flash 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 BIOS loads a copy of that sector into memory at `0x7c00` and then jumps (sets the processor's `%ip`) to that address. When the boot sector begins executing, the processor is simulating an Intel 8088, the core of the original IBM PC released in 1981. The xv6 boot sector's job is to put the processor in a more modern operating mode and then transfer control to the xv6 kernel. In xv6, the boot sector comprises two source files, one written in a combination of 16-bit and 32-bit x86 assembly (`bootasm.S`; (0900)) and one written in C (`bootmain.c`; (1100)). This chapter examines the operation of the xv6 boot sector, from the time the BIOS starts it to the time it transfers control to the kernel proper. The boot sector is a microcosm of the kernel itself: it contains low-level assembly and C code, it manages its own memory, and it even has a device driver, all in under 512 bytes of machine code.

Code: Assembly bootstrap

The first instruction in the boot sector is `cli` (0915), which disables processor interrupts. Interrupts are a way for hardware devices to invoke operating system functions called interrupt handlers. The BIOS is a tiny operating system, and it might have set up its own interrupt handlers as part of the initializing the hardware. But the BIOS isn't running anymore—xv6 is, or will be—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.

Remember that the processor is simulating an Intel 8088. The Intel 8088 had eight 16-bit general-purpose registers but 20 wires in its address bus leading to memory, and thus could be connected to 1 megabyte of memory. The segment registers `%cs`, `%ds`, `%es`, and `%ss` provided the additional bits necessary to generate 20-bit memory addresses from 16-bit registers. In fact, they provide more than enough bits: the segment registers are 16 bits wide too. A full memory reference on the 8088 consists of two 16-bit words, a segment and an offset, written *segment*:*offset*. Typically, the segment is taken from a segment register and the offset from a general-purpose register. For example, the `movs` instruction copies data from `%ds`:`%si` to `%es`:`%di`. The 20-bit memory address that went out on the 8088 bus was the segment times 16 plus the offset. We'll call the addresses the processor chip sends to memory "physical addresses," and the addresses that programs directly manipulate "virtual addresses." Thus, on an

8088, a virtual address consists of a 16-bit segment register combined with a 16-bit general-purpose register, for example `0x8765:0x4321`, and translates to a 20-bit physical address sent to the memory chips, in this case `0x87650+0x4321 = 0x8b971`.

PC BIOSes guarantee to copy the boot sector to physical address `0x7c00` and start it executing, but there is no guarantee that they will choose to set `%cs:%ip` to `0x0000:0x7c00`. In fact, some BIOSes use `0x0000:0x7c00` when the boot sector is from a hard disk and use `0x07c0:0x0000` when the boot sector is from a bootable CD or DVD. There are no guarantees at all about the initial contents of the segment registers used for data accesses (`%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` (0918-0921).

The address calculation can produce a 21-bit address, but the Intel 8088 could only address 20 bits of memory, so it discarded the top bit: `0xffff0+0xffff = 0x10ffef`, but virtual address `0xffff:0xffff` on the 8088 referred to physical address `0x0ffef`. The Intel 80286 had 24-bit physical addresses and thus could address 16 megabytes of memory, so its real mode did not discard the top bit: virtual address `0xffff:0xffff` on the 80286 referred to physical address `0x10ffef`. The IBM PC AT, IBM's 1984 update to the IBM PC, used an 80286 instead of an 8088, but by then there were programs that depended on the 8088's address truncation and did not run correctly on the 80286.

IBM worked around this incompatibility in hardware: the PC AT design connected the 20th address line of memory (A20) to the logical AND of the 20th address line coming out of the 80286 processor and the second bit of the keyboard controller's output port. When the PC AT booted, the keyboard output port's second bit was zero, making the memory controller always see zero on the A20 line, which in turn made the 80286's memory accesses behave like an 8088, so that 8088 programs would run correctly. Of course, IBM wanted to allow new programs to take advantage of the expanded memory. PC AT-specific software instructed the keyboard controller to change the output port bit to a 1, allowing the 80286's A20 values to pass unfiltered to the memory controller. To this day, modern PCs continue this backwards compatibility dance, and low-level software probably continues to depend on 8088 behavior at boot. The boot sector must enable the A20 line using I/O to the keyboard controller on ports 0x64 and 0x60 (0923-0941).

The 8088 had 16-bit general-purpose registers, so that a program that wanted to use more than 65,536 bytes of memory required awkward manipulation of segment registers. The 8088's 20-bit physical addresses also limited the total amount of RAM to a size that seems small today. Modern software expects to be able to use tens to thousands of megabytes of memory, and expects to be able to do it without fiddling with segment registers. The minimum modern expectation is that a processor should have 32-bit registers that can be used directly as addresses. The Intel x86 architecture arrived at those capabilities in two stages of evolution. The 80286 introduced "protected mode" which allowed the segmentation scheme to generate physical addresses with as many bits as required. The 80386 introduced "32-bit mode" which replaced the 80286's 16-bit registers with 32-bit registers. The xv6 boot sequence enables both modes as follows.

In protected mode, a segment register is not a simple base memory address anymore. Instead, it is an index into a segment descriptor table. Each table entry specifies a base physical address, a maximum virtual address called the limit, and permission bits for the segment. These additions are the protection in protected mode: they can be used to make sure that one program cannot access memory belonging to another program (including the operating system itself). Chapter 2 will put the protection features to good use; the boot sector simply wants access to more than 20 bits of memory. It executes an `lgdt` instruction (0954) to set the processor's global descriptor table (GDT) register with the value `gdtdesc` (0995-0997), which in turns points at the table `gdt` (0990-0993).

This simple GDT has three entries: the processor requires entry 0 to be a null entry; entry 1 is a 32-bit code segment with offset 0 and limit 0xffffffff, allowing access to all of physical memory; and entry 2 is a data segment with the same offset and limit. "32-bit code segment" enables the 80386's 32-bit mode, so that the processor will default to 32-bit registers, addresses, and arithmetic when executing in the segment. In protected mode, the bottom two bits of a segment register give the processor's privilege level (0 is kernel, 3 is user mode, 1 and 2 are intermediate). The next bit selects between the global descriptor table (0) and a second table called the local descriptor table (1). The rest of the bits are an index into the given table. Thus 0x8 and 0x10 refer to GDT entries 1 and 2 with kernel privilege level. Those entries are the code and data segments the boot sector will use in protected mode. The code refers to these numbers using the aliases `SEG_KCODE` and `SEG_KDATA` (0907-0908).

Once it has loaded the GDT register, the boot sector enables protected mode, by setting the 1 bit (`CR0_PE`) in register `%cr0` (0955-0957). Enabling protected mode does not change how the processor translates virtual to physical addresses or whether it is in 32-bit mode; 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. Thus the processor continues to execute in 16-bit mode with the same segment translations as before. The switch to 32-bit mode happens when the code executes a far jump (`ljmp`) instruction (0961). The jump continues execution at the next line (0964) but in doing so sets `%cs` to `SEG_KCODE`, which causes the processor to load the descriptor entry from the `gdt` table. The entry describes a 32-bit code segment, so the processor switches into 32-bit mode. The boot sector code has nursed the processor through an evolution from 8088 through 80286 to 80386.

The boot sector's first action in 32-bit mode is to initialize the data segment registers with `SEG_KDATA` (0966-0969). The segments are set up so that the processor uses 32-bit virtual addresses directly as 32-bit physical addresses, without translation, so the software can now conveniently use all of the machine's memory. 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 sector itself is at `0x7c00` through `0x7d00`. Essentially any other section of memory would be a fine location for the stack. The boot sector 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 sector code.

Finally the boot sector calls the C function `bootmain` (0976). `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 (0978-0984). On real hardware, there is no device connected to that port, so this code does nothing. If the boot sector is running inside the PC simulator Bochs, port 0x8a00 is connected to Bochs itself; the code sequence triggers a Bochs debugger breakpoint. Bochs or not, the code then executes an infinite loop (0985-0986). A real boot sector might attempt to print an error message first.

Code: C bootstrap

The C part of the boot sector, bootmain.c (1100), loads a kernel from an IDE disk into memory and then starts executing it. The kernel is an ELF format binary, defined in elf.h. An ELF binary is an ELF file header, struct elfhdr (0855), followed by a sequence of program section headers, struct proghdr (0874). Each proghdr describes a section of the kernel that must be loaded into memory. These headers typically take up the first hundred or so bytes of the binary. To get access to the headers, bootmain loads the first 4096 bytes of the file, a gross overestimation of the amount needed (1113). It places the in-memory copy at address 0x10000, another out-of-the-way memory address.

bootmain casts freely between pointers and integers (1123, 1126, and so on). Programming languages distinguish the two to catch errors, but the underlying processor sees no difference. An operating system must work at the processor's level; occasionally it will need to treat a pointer as an integer or vice versa. C allows these conversions, in contrast to languages like Pascal and Java, precisely because one of the first uses of C was to write an operating system: Unix.

Back in the boot sector, what should be an ELF binary header has been loaded into memory at address 0x10000 (1113). The next step is to check that the first four bytes of the header, the so-called magic number, are the bytes 0x7F, 'E', 'L', 'F', or ELF_MAGIC (0852). All ELF binary headers are required to begin with this magic number as identication. If the ELF header has the right magic number, the boot sector assumes that the binary is well-formed. There are many other sanity checks that a proper ELF loader would do, as we will see in Chapter 9, but the boot sector doesn't have the code space. Checking the magic number guards against simply forgetting to write a kernel to the disk, not against malicious binaries.

An ELF header points at a small number of program headers (proghdrs) describing the sections that make up the running kernel image. Each proghdr gives a virtual address (va), the location where the section's content lies on the disk relative to the start of the ELF header (offset), the number of bytes to load from the file (filesz), and the number of bytes to allocate in memory (memsz). If memsz is larger than filesz, the bytes not loaded from the file are to be zeroed. This is more efficient, both in space and I/O, than storing the zeroed bytes directly in the binary. As an example, the xv6 kernel has two loadable program sections, code and data:

```
# objdump -p kernel

kernel:     file format elf32-i386

Program Header:
LOAD off    0x00001000 vaddr 0x00100000 paddr 0x00100000 align 2**12
   filesz 0x000063ca memsz 0x000063ca flags r-x
LOAD off    0x000073e0 vaddr 0x001073e0 paddr 0x001073e0 align 2**12
   filesz 0x0000079e memsz 0x000067e4 flags rw-
 STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
   filesz 0x00000000 memsz 0x00000000 flags rwx
```

Notice that the second section, the data section, has a `memsz` larger than its `filesz`: the first `0x79e` bytes are loaded from the kernel binary and the remaining `0x6046` bytes are zeroed.

`Bootmain` uses the addresses in the `proghdr` to direct the loading of the kernel. It reads each section's content starting from the disk location `offset` bytes after the start of the ELF header, and writes to memory starting at address `va`. `Bootmain` calls `readseg` to load data from disk (1137) and calls `stosb` to zero the remainder of the segment (1139). `Stosb` (0442) uses the x86 instruction `rep stosb` to initialize every byte of a block of memory.

`Readseg` (1179) reads at least `count` bytes from the disk `offset` into memory at `va`. The x86 IDE disk interface operates in terms of 512-byte chunks called sectors, so `readseg` may read not only the desired section of memory but also some bytes before and after, depending on alignment. For the second program segment in the example above, the boot sector will call `readseg((uchar*)0x1073e0, 0x73e0, 0x79e)`. Due to sector granularity, this call is equivalent to `readseg((uchar*)0x107200, 0x7200, 0xa00)`: it reads `0x1e0` bytes before the desired memory region and `0x82` bytes afterward. In practice, this sloppy behavior turns out not to be a problem (see exercise XXX). `Readseg` begins by computing the ending virtual address, the first memory address above `va` that doesn't need to be loaded from disk (1183), and rounding `va` down to a sector-aligned disk offset . Then it converts the offset from a byte offset to a sector offset; it adds 1 because the kernel starts at disk sector 1 (disk sector 0 is the boot sector). Finally, it calls `readsect` to read each sector into memory.

`Readsect` (1160) reads a single disk sector. It is our first example of a device driver, albeit a tiny one. `Readsect` begins by calling `waitdisk` to wait until the disk signals that it is ready to accept a command. The disk does so by setting the top two bits of its status byte (connected to input port `0x1f7`) to `01`. `Waitdisk` (1151) reads the status byte until the bits are set that way. Chapter 6 will examine more efficient ways to wait for hardware status changes, but busy waiting like this (also called polling) is fine for the boot sector.

Once the disk is ready, `readsect` issues a read command. It first writes command arguments—the sector count and the sector number (offset)—to the disk registers on output ports `0x1f2-0x1f6` (1164-1168). The bits `0xe0` in the write to port `0x1f6` signal to the disk that `0x1f3-0x1f6` contain a sector number (a so-called linear block address), in contrast to a more complicated cylinder/head/sector address used in early PC disks. After writing the arguments, `readsect` writes to the command register to trigger the read (1154). The command `0x20` is "read sectors." Now the disk will read the data stored in the specified sectors and make it available in 32-bit pieces on input port `0x1f0`. `Waitdisk` (1151) waits until the disk signals that the data is ready, and then the call to `insl` reads the 128 (`SECTSIZE/4`) 32-bit pieces into memory starting at `dst` (1173).

`Inb`, `outb`, and `insl` are not ordinary C functions. They are inlined functions whose bodies are assembly language fragments (0403, 0421, 0412). When gcc sees the call to `inb` (1154), the inlined assembly causes it to emit a single `inb` instruction. This style allows the use of low-level instructions like `inb` and `outb` while still writing the control logic in C instead of assembly.

The implementation of `insl` (0412) is worth looking at more closely. `Rep insl` is actually a tight loop masquerading as a single instruction. The `rep` prefix executes the following instruction `%ecx` times, decrementing `%ecx` after each iteration. The `insl` instruction reads a 32-bit value from port `%dx` into memory at address `%edi` and then increments `%edi` by 4. Thus `rep insl` copies 4×`%ecx` bytes, in 32-bit chunks, from port `%dx` into memory starting at address `%edi`. The register annotations tell GCC to prepare for the assembly sequence by storing `dst` in `%edi`, `cnt` in `%ecx`, and `port` in `%dx`. Thus the `insl` function copies 4×`cnt` bytes from the 32-bit port `port` into memory starting at `dst`. The `cld` instruction clears the processor's direction flag, so that the `insl` instruction increments `%edi`; when the flag is set, `insl` decrements `%edi` instead. The x86 calling convention does not define the state of the direction flag on entry to a function, so each use of an instruction like `insl` must initialize it to the desired value.

The boot loader is almost done. `Bootmain` loops calling `readseg`, which loops calling `readsect` (1135-1140). At the end of the loop, `bootmain` has loaded the kernel into memory. Now it is time to run the kernel. The ELF header specifies the kernel entry point, the `%eip` where the kernel expects to be started (just as the boot loader expected to be started at `0x7c00`). `Bootmain` casts the entry point integer to a function pointer and calls that function, essentially jumping to the kernel's entry point (1144-1145). The kernel should not return, but if it does, `bootmain` will return, and then `bootasm.S` will attempt a Bochs breakpoint and then loop forever.

Where is the kernel in memory? `Bootmain` does not directly decide; it just follows the directions in the ELF headers. The "linker" creates the ELF headers, and the xv6 Makefile that calls the linker tells it that the kernel should start at `0x100000`.

Assuming all has gone well, the kernel entry pointer will be the kernel's `main` function (see `main.c`). The next chapter continues there.

Real world

The boot sector described in this chapter 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 sector 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 sector like the one in this chapter 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.

TODO: Also, x86 does not imply BIOS: Macs use EFI. I wonder if the Mac has an A20 line.

Exercises

1. Look at the kernel load addresses; why doesn't the sloppy readsect cause problems?
2. something about BIOS lasting longer + security problems
3. Suppose you wanted bootmain() to load the kernel at 0x200000 instead of 0x100000, and you did so by modifying bootmain() to add 0x100000 to the va of each ELF section. Something would go wrong. What?

Chapter 2

Processes

One of an operating system's central roles is to allow multiple programs to share the CPUs and main memory safely, isolating them so that one errant program cannot break others. To that end, xv6 provides the concept of a process, as described in Chapter 0. xv6 implements a process as a set of data structures, but a process is quite special: it comes alive with help from the hardware. This chapter examines how xv6 allocates memory to hold process code and data, how it creates a new process, and how it configures the processor's segmentation hardware to give each process the illusion that it hash its own private memory address space. The next few chapters will examine how xv6 uses hardware support for interrupts and context switching to create the illusion that each process has its own private CPU.

Code: Memory allocation

xv6 allocates most of its data structures statically, by declaring C global variables and arrays. The linker and the boot loader cooperate to decide exactly what memory locations will hold these variables, so that the C code doesn't have to explicitly allocate memory. However, xv6 does explicitly and dynamically allocate physical memory for user process memory, for the kernel stacks of user processes, and for pipe buffers. When xv6 needs memory for one of these purposes, it calls `kalloc`; when it no longer needs them memory, it calls `kfree` to release the memory back to the allocator. Xv6's memory allocator manages blocks of memory that are a multiple of 4096 bytes, because the allocator is used mainly to allocate process address spaces, and the x86 segmentation hardware manages those address spaces in multiples of 4 kilobytes. The xv6 allocator calls one of these 4096-byte units a page, though it has nothing to do with paging.

`Main` calls `kinit` to initialize the allocator (1226). `Kinit` ought to begin by determining how much physical memory is available, but this turns out to be difficult on the x86. Xv6 doesn't need much memory, so it assumes that there is at least one megabyte available past the end of the loaded kernel and uses that megabyte. The kernel is around 50 kilobytes and is loaded one megabyte into the address space, so xv6 is assuming that the machine has at least a little more than two megabytes of memory, a very safe assumption on modern hardware.

`Kinit` (2277) uses the special linker-defined symbol `end` to find the end of the kernel's static data and rounds that address up to a multiple of 4096 bytes (2284). When `n` is a power of two, the expression `(a+n-1) & ~(n-1)` is a common C idiom to round `a` up to the next multiple of `n`. `Kinit` then does a surprising thing: it calls `kfree` to free a megabyte of memory starting at that address (2287). The discussion of `kalloc` and `kfree` above said that `kfree` was for returning memory allocated with `kalloc`, but that was a client-centric perspective. From the allocator's point of view, calls to `kfree` give it memory to hand out, and then calls to `kalloc` ask for the memory back. The allocator starts with no memory; this initial call to `kfree` gives it a megabyte to manage.

The allocator maintains a *free list* of memory regions that are available for allocation. It keeps the list sorted in increasing order of address in order to ease the task of merging contiguous blocks of freed memory. Each contiguous region of available memory is represented by a `struct run`. But where does the allocator get the memory to hold that data structure? The allocator does another surprising thing: it uses the memory being tracked as the place to store the `run` structure tracking it. Each `run *r` represents the memory from address `(uint)r` to `(uint)r + r->len`. The free list is protected by a spin lock (2262-2265). 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.

`Kfree` (2305) begins by setting every byte in the memory being freed to the value 1. This step is unnecessary for correct operation, but it helps break incorrect code that continues to refer to memory after freeing it. This kind of bug is called a dangling reference. By setting the memory to a bad value, `kfree` increases the chance of making such code use an integer or pointer that is out of range (`0x01010101` is around 16 million).

`Kfree`'s first real work is to store a `run` in the memory at `v`. It uses a cast in order to make `p`, which is a pointer to a `run`, refer to the same memory as `v`. It also sets `pend` to the `run` for the block following `v` (2316-2317). If that block is free, `pend` will appear in the free list. Now `kfree` walks the free list, considering each run `r`. The list is sorted in increasing address order, so the new run `p` belongs before the first run `r` in the list such that `r> pend`. The walk stops when either such an `r` is found or the list ends, and then `kfree` inserts `p` in the list before `r` (2337-2340). The odd-looking `for` loop is explained by the assignment `*rp = p`: in order to be able to insert `p` *before* `r`, the code had to keep track of where it found the pointer `r`, so that it could replace that pointer with `p`. The value `rp` points at where `r` came from.

There are two other cases besides simply adding `p` to the list. If the new run `p` abuts an existing run, those runs need to be coalesced into one large run, so that allocating and freeing small blocks now does not preclude allocating large blocks later. The body of the `for` loop checks for these conditions. First, if `rend == p` (kalloc.c/rend.==.p/), then the run `r` ends where the new run `p` begins. In this case, `p` can be absorbed into `r` by increasing `r`'s length. If growing `r` makes it abut the next block in the list, that block can be absorbed too (kalloc.c/r->next && r->next == pend/,/}/). Second, if `pend == r` (kalloc.c/pend.==.r/), then the run `p` ends where the new run `r` begins. In this case, `r` can be absorbed into `p` by increasing `p`'s length and then replacing `r` in the list with `p` (2330-2335).

`Kalloc` has a simpler job than `kfree`: it walks the free list looking for a run that is large enough to accommodate the allocation. When it finds one, `kalloc` takes the memory from the end of the run (2364-2365). If the run has no memory left, `kalloc` deletes the run from the list (2367-2368) before returning.

Code: Process creation

This section describes how xv6 creates the very first process. Xv6 represents each process by a `struct proc` (1529) entry in the statically-sized `ptable.proc` process table. The most important fields of a `struct proc` are `mem`, which points to the physical memory containing the process's instructions, data, and stack; `kstack`, which points to the process's kernel stack for use in interrupts and system calls; and and `state`, which indicates whether the process is allocated, ready to run, running, etc.

The story of the creation of the first process starts when `main` (1235) calls `userinit` (1802), whose first action is to call `allocproc`. The job of `allocproc` (1754) is to allocate a slot in the process table and to initialize the parts of the process's state required for it to execute in the kernel. `Allocproc` is called for all new processes, while `userinit` is only called for the very first process. `Allocproc` scans the table for a process with state `UNUSED` (1669-1762). When it finds an unused process, `allocproc` sets the state to `EMBRYO` to mark it as used and gives the processes a unique `pid` (1658-1768). Next, it tries to allocate a kernel stack for the process. If the memory allocation fails, `allocproc` changes the state back to `UNUSED` and returns zero to signal failure.

Now `allocproc` must set up the new process's kernel stack. As we will see in Chapter 3, the usual way that a process enters the kernel is via an interrupt mechanism, which is used by system calls, interrupts, and exceptions. The process's kernel stack is the one it uses when executing in the kernel during the handling of that interrupt. `Allocproc` 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, so that the ordinary code for returning from the kernel back to the user part of a process will work. These values are a `struct trapframe` which stores the user registers, the address of the kernel code that returns from an interrupt (`trapret`) for use as a function call return address, and a `struct context` which holds the process's kernel registers. When the kernel switches contexts to this new process, the context switch will restore its kernel registers; it will then execute kernel code to return from an interrupt and thus restore the user registers, and then execute user instructions. `Allocproc` sets `p->context->eip` to `forkret`, so that the process will start executing in the kernel at the start of `forkret`. The context switching code will start executing the new process with the stack pointer set to `p->context+1`, which points to the stack slot holding the address of the `trapret` function, just as if `forkret` had been called by `trapret`.

```
 ----------  <-- top of new process's kernel stack
| esp      |
| ...      |
| eip      |
| ...      |
| edi      | <-- p->tf (new proc's user registers)
| trapret  | <-- address forkret will return to
| eip      |
| ...      |
| edi      | <-- p->context (new proc's kernel registers)
```

```
|          |
| (empty)  |
|          |
 ----------  <-- p->kstack
```

Main calls userinit to create the first user process (1235). Userinit (1802) calls allocproc, saves a pointer to the process as initproc, ad then configures the new process's user state. First, the process needs memory. This first process is going to execute a very tiny program (initcode.S; (6700)), so the memory need only be a single page (1811-1812). The initial contents of that memory are the compiled form of initcode.S; as part of the kernel build process, the linker embeds that binary in the kernel and defines two special symbols _binary_initcode_start and _binary_initcode_size telling the location and size of the binary (XXX sidebar about why it is extern char[]). Userinit copies that binary into the new process's memory and zeros the rest (1813-1814). Then it sets up the trap frame with the initial user mode state: the cs register contains a segment selector for the SEG_UCODE segment running at privilege level DPL_USER (i.e., user mode not kernel mode), and similarly ds, es, and ss use SEG_UDATA with privilege DPL_USER. The eflags FL_IF is set to allow hardware interrupts; we will reexamine this in Chapter 3. The stack pointer esp is the process's largest valid virtual address, p->sz. The instruction pointer is the entry point for the initcode, address 0. Note that initcode is not an ELF binary and has no ELF header. It is just a small headerless binary that expects to run at address 0, just as the boot sector is a small headerless binary that expects to run at address 0x7c00. Userinit sets p->name to initcode mainly for debugging. Setting p->cwd sets the process's current working directory; we will examine namei in detail in Chapter 7.

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

Code: Running a process

Rather than use special code to start the first process running and guide it to user space, xv6 has chosen to set up the initial data structure state as if that process was already running. But it wasn't running and still isn't: so far, this has been just an elaborate construction exercise, like lining up dominoes. Now it is time to knock over the first domino, set the operating system and the hardware in motion and watch what happens.

Main calls ksegment to initialize the kernel's segment descriptor table (1219). Ksegment initializes a per-CPU global descriptor table c->gdt with the same segments that the boot sector configured (and one more, SEG_KCPU, which we will revisit in Chapter 4). After calling userinit, which we examined above, main calls scheduler to start running user processes (1263). Scheduler (1908) looks for a process with p->state set to RUNNABLE, and there's only one it can find: initproc. It sets the global variable cp to the process it found (cp stands for current process) and calls usegment to create segments on this CPU for the user-space execution of the process (1846). Usegment (1722) creates code and data segments SEG_UCODE and SEG_UDATA mapping addresses 0 through cp->sz-1 to the memory at cp->mem. It also creates a new task state segment SEG_TSS that instructs the hardware to handle an interrupt by returning to kernel mode with ss and esp set to SEG_KDATA<<3 and (uint)cp->kstack+KSTACKSIZE, the top of this process's kernel stack. We will reexamine the task state segment in Chapter 3.

Now that usegment has created the user code and data segments, the scheduler can start running the process. It sets p->state to RUNNING and calls swtch (2208), to perform a context switch from one kernel process to another; in this invocation, from a scheduler process to p. Swtch, which we will reexamine in Chapter 5, saves the scheduler's registers that must be saved; i.e., the context (1518) that a process needs to later resume correctly. Then, Swtch loads p->context into the hardware registers. The final ret instruction (2227) pops a new eip from the stack, finishing the context switch. Now the processor is running process p.

Allocproc set initproc's p->context->eip to forkret, so the ret starts executing forkret. Forkret (1984) releases the ptable.lock (see Chapter 4) and then 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 (2529) uses pop instructions to walk up the trap frame just as swtch did with the kernel context: popal restores the general registers, then the popl instructions restore %gs, %fs, %es, and %ds. The addl skips over the two fields trapno and errcode. Finally, the iret instructions pops %cs, %eip, and %eflags off the stack. The contents of the trap frame have been transferred to the CPU state, so the processor continues at the %cs:%eip specified in the trap frame. For initproc, that means SEG_UCODE:0, the first instruction of initcode.S.

At this point, %eip holds zero and %esp holds 4096. These are virtual addresses in the process's user address space. The processor's segmentation machinery translates them into physical addresses. The relevant segmentation registers (cs, ds, and ss) and segment descriptors were set up by userinit and usegment to translate virtual address zero to physical address p->mem, with a maximum virtual address of p->sz. The fact that the process is running with CPL=3 (in the low bits of cs) means that it cannot use the segment descriptors SEG_KCODE and SEG_KDATA, which would give it access to all of physical memory. So the process is constrained to using only its own memory.

Initcode.S (6707) 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 (6720-6722). If the exec fails and does return, initcode loops calling the exit system call, which definitely should not return (6714-6718).

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 reuses code that xv6 must provide for standard operation.

The next chapter examines how xv6 configures the x86 hardware to handle the system call interrupt caused by int $T_SYSCALL. The rest of the book builds up enough of the process management and file system implementation to finally implement `exec` in Chapter 9.

Real world

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

Xv6 departs from modern operating systems in its use of segmentation registers for process isolation and address translation. Most operating systems for the x86 uses the paging hardware for address translation and protection; they treat the segmentation hardware mostly as a nuisance to be disabled by creating no-op segments like the boot sector did. However, a simple paging scheme is somewhat more complex to implement than a simple segmentation scheme. Since xv6 does not aspire to any of the advanced features which would require paging, it uses segmentation instead.

The one common use of segmentation is to implement variables like xv6's `cp` that are at a fixed address but have different values in different threads. Implementations of per-CPU (or per-thread) storage on other architectures would dedicate a register to holding a pointer to the per-CPU data area, but the x86 has so few general registers that the extra effort required to use segmentation is worthwhile.

xv6's use of segmentation instead of paging is awkward in a couple of ways, even given its low ambitions. First, it causes user-space address zero to be a valid address, so that programs do not fault when they dereference null pointers; a paging system could force faults by marking the first page invalid, which turns out to be invaluable for catching bugs in C code. Second, xv6's segment scheme places the stack at a relatively low address which prevents automatic stack extension. Finally, all of a process's memory must be contiguous in physical memory, leading to fragmentation and/or copying.

In the earliest days of operating systems, each operating system was tailored to a specific hardware configuration, so the amount of memory could be a hard-wired constant. As operating systems and machines became commonplace, most developed a way to determine the amount of memory in a system at boot time. 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. None of these is guaranteed to be reliable, so modern x86 operating systems typically augment one or more of them with complex sanity checks and heuristics. In the interest of simplicity, xv6 assumes that the machine it runs on has at least one megabyte of memory past the end of the kernel. Since the kernel is around 50 kilobytes and is loaded one megabyte into the address space, xv6 is assuming that the machine has at least a little more than 2 MB of memory. A real operating system would have to do a better job.

Memory allocation was a hot topic a long time ago. Basic problem was how to make the most efficient use of the available memory and how best to prepare for future requests without knowing what the future requests were going to be. See Knuth. Today, more effort is spent on making memory allocators fast rather than on making them space-efficient. The runtimes of today's modern programming languages allocate mostly many small blocks. Xv6 avoids smaller than a page allocations by using fixed-size data structures. A real kernel allocator would need to handle small allocations as well as large ones, although the paging hardware might keep it from needing to handle objects larger than a page.

Exercises

1. Set a breakpoint at swtch. Single step through to forkret. Set another breakpoint at forkret's ret. Continue past the release. Single step into trapret and then all the way to the iret. Set a breakpoint at 0x1b:0 and continue. Sure enough you end up at initcode.

2. Do the same thing except single step past the iret. You don't end up at 0x1b:0. What happened? Explain it. Peek ahead to the next chapter if necessary.

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

Chapter 3

System calls, exceptions, and interrupts

An operating system must handle 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. *Exceptions* are illegal program actions that generate an interrupt. Examples of illegal programs actions include divide by zero, attempt to access memory outside segment bounds, and so on. *Interrupts* are generated by hardware devices that need attention of the operating system. For example, a clock chip may generate an interrupt every 100 msec to allow the kernel to implement time sharing. As another example, when the disk has read a block from disk, it generates an interrupt to alert the operating system that the block is ready to be retrieved.

In all three cases, the operating system design must range for the following to happen. The system must save user state for future transparent resume. The system must be set up for continued execution in the kernel. The system must chose a place for the kernel to start executing. The kernel must be able to retrieve information about the event, including arguments. It must all be done securely; the system must maintain isolation of user processes and the kernel.

To achieve this goal the operating system must be aware of the details of how the hardware handles system calls, exceptions, and interrupts. In most processors these three events are handled by a single hardware mechanism. For example, on the x86, a program invokes a system call by generating an interrupt using the `int` instruction. Similarly, exceptions generate an interrupt too. Thus, if the operating system has a plan for interrupt handling, then the operating system can handle system calls and exceptions too.

The basic plan is as follows. An interrupts stops the normal processor loop—read an instruction, advance the program counter, execute the instruction, repeat—and starts executing a new sequence called an interrupt handler. Before starting the interrupt handler, the processor saves its previous state, so that the interrupt handler can restore that state if appropriate.

A challenge in the transition to and from the interrupt handler is that the processor should switch from user mode to kernel mode, and back. If a device generates an interrupt (e.g., the clock chip) and a processor is running a user processor, then we would like to arrange that the kernel handles the interrupt, so that it can switch the processor to a different user process, if the clock interrupt signals the end of the time slice for the current running process. We want this interrupt to be handled by the kernel, because a user program may ignore it so that it doesn't have to give up the processor.

A word on terminology: Although the official x86 term is interrupt, x86 refers to all of these as traps, largely because it was the term used by the PDP11/40 and therefore is the conventional Unix term. This chapter uses the terms trap and interrupt interchangeably, but it is important to remember that traps pertain to the current process running on a processor (e.g., the process makes a system call and as a result generates a trap), and interrupts pertain to devices and may have no relation to the program running on the processor when the interrupts occurs. For example, a disk may generate an interrupt when it is done retrieving a block for a process that is currently not running on any processor because the kernel descheduled it to run another process while the process was waiting for the disk. This property of interrupts makes thinking about interrupts more difficult than thinking about traps, because interrupts happen concurrently with other activities, and requires the designer to think about parallelism and concurrency. A topic that we will address in Chapter 4.

This chapter examines the xv6 trap handlers, covering hardware interrupts, software exceptions, and system calls.

Code: The first system call

The last chapter ended with `initcode.S` invoke a system call. Let's look at that again (6712). Remember from Chapter 2 that the process pushed the arguments for an `exec` call on the process's stack, and system call number in `%eax`. The system call numbers match the entries in the syscalls array, a table of function pointers (2850). We need to arrange that the `int` instruction switches the processor from user space to kernel space, that the kernel invokes the right kernel function (i.e., `sys_exec`), and that the kernel can retrieve the arguments for `sys_exec`. The next few subsections describes how xv6 arranges this for system calls, and then we will discover that we can reuse the same code for interrupts and exceptions.

Code: Assembly trap handlers

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

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

`Tvinit` (2566), called from `main`, sets up the 256 entries in the table `idt`. Interrupt `i` is handled by the `%eip` `vectors[i]`. Each entry point is different, because the x86 provides does not provide the trap number to the interrupt handler. Using 256 different handlers is the only way to distinguish the 256 cases.

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

The kernel also sets for the system call gate the privilege to DPL_USER, which allows a user program to generate the trap with an explicit int instruction. If xv6 didn't set the privilege, then if the user program would invoke int, the processor would generate 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 who knows if the stack is a valid one. The user process may have been malicious or contain an error, and supplied a value in esp, which doesn't correspond to a stack. 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 an stack segment selector and a new value for %esp. IN usegment (1722). has stored the address of the top of the kernel stack of the user process into the task segment descriptor, and the x86 will be load that address in %esp on a trap.

The 256 different handlers must behave differently: for some traps, the x86 pushes an extra error code on the stack, but for most it doesn't. The handlers for the traps without error codes push a fake one on the stack explicitly, to make the stack layout uniform. Instead of writing 256 different functions by hand, we use a Perl script (2450) to generate the entry points. Each entry pushes an error code if the processor didn't, pushes the interrupt number, and then jumps to alltraps, a common body.

Alltraps (2506) continues to save processor state: it pushes %ds, %es, %fs, %gs, and the general-purpose registers (2507-2512). The result of this effort is that the kernel stack now contains a struct trapframe (0552) describing the precise user mode processor state at the time of the trap. The processor pushed cs, eip, and eflags. The processor or the trap vector pushed an error number, and alltraps pushed the rest. The trap frame contains all the information necessary to restore the user mode processor state when the trap handler is done, so that the processor can continue exactly as it was when the trap started.

In the case of the first system call, the saved eip will the address of the instruction right after the int instruction. cs is the user code segment selector. eflags is the content of the eflags register at the point of executing the int instruction. As part of saving the general-purpose registers, alltraps also saved %eax, which contains the system call number, and now that number is also in the trapframe on the kernel stack.

Now that the user mode processor state is saved, alltraps can finishing setting up processor for running kernel code. The processor set the selectors %cs and %ss before entering the handler; alltraps must set %ds and %es (2515-2517). It also sets %fs and %gs to point at the SEG_KCPU per-CPU data segment (2518-2520). Chapter 2 will revisit that segment.

Once the segments are set properly, alltraps can call the C trap handler trap. It pushes %esp, which points at the trap frame we just constructed, onto the stack as an argument to trap (2523). Then it calls trap (2524). After trap returns, alltraps pops the argument off the stack by adding to the stack pointer (2525) 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 register state and then iret jumps back into user space.

The discussion so far has talked about trap saving the user mode processor state, but traps can happen while the kernel is executing too. The same code runs; the only difference is that the saved `%cs`, `%eip`, `%esp`, and segment registers are all kernel values. When the final `iret` restores a kernel mode `%cs`, the processor continues executing in kernel mode.

Code: C trap handler

We saw in the last section that each handler sets up a trap frame and then calls the C function `trap`. `Trap` (2601) looks at the hardware trap number `tf->trapno` to decide why it has been called and what needs to be done. If the trap is `T_SYSCALL`, `trap` calls the system call handler `syscall`. We'll revisit the two `cp->killed` checks in Chapter 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`.

[[Sidebar about panic: panic is the kernel's last resort: the impossible has happened and the kernel does not know how to proceed. In xv6, panic does ...]]

Code: System calls

For system calls, trap invokes `syscall` (2874). `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 9, and `syscall` will invoke the 9th 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 (2880). 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 mechanism left: finding the system call arguments. The helper functions argint and argptr, argstr retrieve the *n*'th system call argument, as either an integer, pointer, or a string. Argint uses the user-space esp register to locate the *n'th* argument: esp points at the return address for the system cal stub. The arguments are right above it, at esp+4. Then the nth argument is at esp + 4 + 4 * *n*.

Argint calls fetchint to read the value at that address from user memory and write it to *ip. Fetchint cannot simply cast the address to a pointer, because kernel and user pointers have different meaning: in the kernel, address 0 means physical address zero, the first location in physical memory. When a user process is executing, the kernel sets the segmentation hardware so that user address zero corresponds to the process's private memory, kernel address p->mem. The kernel also uses the segmentation hardware to make sure that the process cannot access memory outside its local private memory: if a user program tries to read or write memory at an address of p->sz or above, the processor will cause a segmentation trap, and trap will kill the process, as we saw above. Now though, the kernel is running and must implement the memory translation and checks itself. Fetchint checks that the user address is in range and then convert it to a kernel pointer by adding p->mem before reading the value.

Argptr is similar in purpose to argint: it interprets the *n*th system call argument as a user pointer and sets *p to the equivalent kernel pointer. Argptr calls argint to fetch the argument as an integer and then users the same logic as fetchint to interpret the integer as a user pointer and compute the equivalent kernel pointer. Note that two translations occur during a call to argptr. First, the user stack pointer is translated during the fetching of the argument. Then the argument, itself a user pointer, is translated to produce a kernel pointer.

Argstr is the final member of the system call argument trio. It interprets the nth argument as a pointer, like argptr does, but then also ensures that the pointer points at a NUL-terminated string: the NUL must be present before the address space ends.

The system call implementations (for example, sysproc.c and sysfile.c) are typically wrappers: they decode the arguments using argint, argptr, and argstr and then call the real implementations.

Let's look at how `sys_exec` uses these functions to get at its arguments. (to be written.)

Code: Interrupts

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

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

Let's look at the timer device and timer interrupts. We would like the timer hardware to generate an interrupt, say, 100 times per second so that the kernel can track the passage of time and so the kernel can time-slice among multiple running processes. The choice of 100 times per second allows for decent interactive performance while not swamping the processor with handling interrupts.

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

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

To also work correctly on uniprocessors, Xv6 programs the programmable interrupt controler (PIC) (5982). Each PIC can handle a maximum of 8 interrupts (i.e., devices) and multiplex them on the interrupt pin of the processor. To allow for more than 8 devices, PICs can be cascaded and typically boards have at least two. Using `inb` and `outb` instructions Xv6 programs the master to generate IRQ 0 through 7 and the slave to generate IRQ 8 through 16. Initially xv6 programs the PIC to mask all interrupts. The code in `timer.c` sets timer 1 and enables the timer interrupt on the PIC (6674). This description omits some of the details of programming the PIC. These details of the PIC (and the IOAPIC and LAPIC) are not important to this text but the interested reader can consult the manuals for each device, which are referenced in the source files.

On multiprocessors, xv6 must program the IOAPIC, and the LAPIC on each processor. The IO APIC has a table and the processor can program entries in the table through memory-mapped I/O, instead of using `inb` and `outb` instructions. During initialization, xv6 programs to map interrupt 0 to IRQ 0, and so on, but disables them all. Specific devices enable particular interrupts and say to which processor the interrupt should be routed. For example, xv6 routes keyboard interrupts to processor 0 (6616). Xv6 routes disk interrupts to the highest numbered processor on the system (3351).

The timer chip is inside the LAPIC, so that each processor can receive timer interrupts independently. Xv6 sets it up in `lapicinit` (5701). The key line is the one that programs the timer (5714). This line tells the LAPIC to periodically generate an interrupt at `IRQ_TIMER`, which is IRQ 0. Line (5743) 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` flags 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 (0915) and the other processors (1029). The scheduler on each processor enables interrupts (1650). To control that certain code fragments are not interrupted, xv6 disables interrupts during these code fragments (e.g., see `usegment` (1722)).

The timer interrupts through vector 32 (which xv6 chose to handle IRQ 0), which xv6 setup in `idtinit` (1259). 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. Interrupt gates clears `IF`, so that the interrupted processor doesn't receive interrupts while it is handling the current interrupt. From here on until `trap`, interrupts follow the same code path as system calls and exceptions, building up a trap frame.

`Trap` when it's called for a time interrupt, does just two things: increment the ticks variable (2562), and call `wakeup`. The latter, as we will see in Chapter 5, may cause the interrupt to return in a different process.

Real world

polling

memory-mapped I/O versus I/O instructions

interrupt handler (trap) table driven.

Interrupt masks. Interrupt routing. On multiprocessor, different hardware but same effect.

interrupts can move.

more complicated routing.

more system calls.

have to copy system call strings.

even harder if memory space can be adjusted.

Supporting all the devices on a PC motherboard in its full glory is much work, because the drivers to manage the devices can get complex.

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: locks. Locks provide mutual exclusion, ensuring that only one CPU at a time can hold a 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.

As an example, consider the implementation of a simple linked list:

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

Proving this implementation correct is a typical exercise in a data structures and algorithms class. Even though this implementation can be proved correct, it isn't, at least not on a multiprocessor. If two different CPUs execute `insert` at the same time, it could happen that both execute line 15 before either executes 16. 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 are consequently difficult to reproduce. For example, adding print statements while debugging `insert` might change the timing of the execution enough to make the race disappear.

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

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

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

Code: Locks

Xv6's represents a lock as a `struct spinlock` (1301). 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

```
void
acquire(struct spinlock *lk)
{
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
```

```
29        }
30    }
```

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

To execute those two lines atomically, xv6 relies on a special 386 hardware instruction, `xchg` (0501). In one atomic operation, `xchg` swaps a word in memory with the contents of a register. `Acquire` (1373) repeats this `xchg` instruction in a loop; each iteration reads `lk->locked` and atomically sets it to 1 (1382). If the lock is held, `lk->locked` will already be 1, so the `xchg` returns 1 and the loop continues. If the `xchg` returns 0, however, `acquire` has successfully acquired the lock—`locked` was 0 and is now 1—so the loop can stop. Once the lock is acquired, `acquire` records, for debugging, the CPU and stack trace that acquired the lock. When a process acquires a lock and forget to release it, this information can help to identify the culprit. These debugging fields are protected by the lock and must only be edited while holding the lock.

`Release` (1402) is the opposite of `acquire`: it clears the debugging fields and then releases the lock.

Modularity and recursive locks

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

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

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

Code: Using locks

The hardest 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, remeber 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. For protecting kernel data structures, it would suffice to create a single lock that must be acquired on entering the kernel and released on exiting the kernel. Many uniprocessor operating systems have been converted to run on multiprocessors using this approach, sometimes called a "giant kernel lock," but the approach sacrifices true concurrency: only one CPU can execute in the kernel at a time. If the kernel does any heavy computation, it would be more efficient to use a larger set of more fine-grained locks, so that the kernel could execute on multiple CPUs simultaneously.

Ultimately, the choice of lock granularity is more art than science. Xv6 uses a few coarse data-structure specific locks. Hopefully, the examples of xv6 will help convey a feeling for some of the art.

Chapter 5

Scheduling

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

Xv6 has provides this plan. If two different processes are competing for a single CPU, xv6 multiplexes them, switching many times per second between executing one and the other. Xv6 uses multiplexing to create the illusion that each process has its own CPU, just as xv6 used the memory allocator and hardware segmentation to create the illusion that each process has its own memory.

Implementing multiplexing has a few challenges. First, how to switch from process to another? Xv6 uses the standard mechanism of context switching; although the idea is simple, the code to implement is typically among the most opaque code in an operating system. Second, how to do context switching transparently? Xv6 uses the standard technique to force context switch in the timer interrupt handler, Third, may processes may be switching concurrently, and a locking plan is necessary to avoid races. Fourth, when a process completed its execution, it shouldn't be multiplexed with other processes, but cleaning a process is not easy; it cannot clean up itself since that requires that it runs. Xv6 tries to solve these problems as straightforward as possible, but nevertheless the resulting code is tricky.

Once there are multiple processes executing, xv6 must also provide some way for them to coordinate among themselves. Often it is necessary for one process to wait for another to perform some action. Rather than make the waiting process waste CPU by repeatedly checking whether that action has happened, xv6 allows a process to sleep waiting for an event and allows another process to wake the first process. Because processes run in parallel, there is a risk of losing a wake up. As an example of these problems and their solution, this chapter examines the implementation of pipes.

Code: Scheduler

Chapter 2 breezed through the scheduler on the way to user space. Let's take a closer look at it. Each processor runs `mpmain` at boot time; the last thing `mpmain` does is call `scheduler` (1263).

`Scheduler` (1908) runs a simple loop: find a process to run, run it until it stops, repeat. At the beginning of the loop, `scheduler` enables interrupts with an explicit `sti` (1914), so that if a hardware interrupt is waiting to be handled, the scheduler's CPU will handle it before continuing. Then the scheduler loops over the process table looking for a runnable process, one that has p->state == RUNNABLE. Once it finds a process, it sets the per-CPU current process variable cp, updates the user segments with usegment, marks the process as RUNNING, and then calls swtch to start running it (1922-1928).

Code: Context switching

Every xv6 process has its own kernel stack and register set, as we saw in Chapter 2. Each CPU has its own kernel stack to use when running the scheduler. Swtch saves the scheduler's context—it's stack and registers—and switches to the chosen process's context. When it is time for the process to give up the CPU, it will call swtch to save its own context and return to the scheduler context. Each context is represented by a struct context*, a pointer to a structure stored on the stack involved. Swtch takes two arguments struct context **old and struct context *new; it saves the current context, storing a pointer to it in *old and then restores the context described by new.

Instead of following the scheduler 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 cp->context and switch to the scheduler context previously saved in c->context (1967).

Swtch (2202) starts by loading its arguments off the stack into the registers %eax and %edx (2209-2210); swtch must do this before it changes the stack pointer and can no longer access the arguments via %esp. Then swtch pushes the register state, creating a context structure on the current stack. Only the callee-save registers need to be saved; the convention on the x86 is that these are %ebp, %ebx, %esi, %ebp, and %esp. Swtch pushes the first four explicitly (2213-2216); it saves the last implicitly as the struct context* written to *old (2219). 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 (2220). 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 (2223-2227). Because swtch has changed the stack pointer, the values restored and the address returned to are the ones from the new context.

In our example, sched's called swtch to switch to c->context, the per-CPU scheduler context. That new context had been saved by scheduler's call to swtch (1928). When the swtch we have been tracing returns, it returns not to sched but to scheduler, and its stack pointer points at the scheduler stack, not initproc's kernel stack.

Code: Scheduling

The last section looked at the low-level details of swtch; now let's take swtch as a given and examine the conventions involved in switching from process to scheduler and back to process. The convention in xv6 is that 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 (cp->state), and then call sched. Yield (1973) follows this convention, as do sleep and exit, which we will examine later. Sched double checks those conditions (1957-1962) and then an implication: since a lock is held, the CPU should be running with interrupts disabled. Finally, sched calls swtch to save the current context in cp->context and switch to the scheduler context in c->context. Swtch returns on the scheduler's stack as though scheduler's swtch had returned (1928). The scheduler continues the for loop, finds a process to run, switches to it, and the cycle repeats.

We just saw that xv6 holds ptable.lock across calls to swtch: the caller of swtch must already hold the lock, and control of the lock passes to the switched-to code. This convention is unusual with locks; the typical convention is the thread that acquires a lock is also responsible of releasing the lock, which makes it easier to reason about correctness. For context switching is necessary to break the typical convention because ptable.lock protects the state and context fields in each process structure. Without the lock, it could happen that a process decided to yield, set its state to RUNNABLE, and then before it could swtch to give up the CPU, a different CPU would try to run it using swtch. This other CPU's call to swtch would use a stale context, the one from the last time the process was started, causing time to appear to move backward. It would also cause two CPUs to be executing on the same stack. Both are incorrect.

To avoid this problem, xv6 follows the convention that the thread that releases a processor acquires the ptable.lock lock and the thread that receives that processor next releases the lock. To make this convention clear, a thread gives up its processor always in sched, switches always to the same location in the scheduler thread, which returns a processor always in sched. Thus, if one were to print out the line numbers where xv6 switches threads, one would observe the following simple pattern: (1928), (1967), (1928), (1967), and so on. The procedures in which this stylized switching between two threads happens are sometimes referred to as co-routines; in this example, sched and scheduler are co-routines of each other.

There is one case when the scheduler's swtch to a new process does not end up in sched. We saw this case in Chapter 2: when a new process is first scheduled, it begins at forkret (1984). Forkret exists only to honor this convention by releasing the ptable.lock; otherwise, the new process could start at trapret.

Sleep and wakeup

Locks help CPUs and processes avoid interfering with each other, and scheduling help processes share a CPU, but so far we have no abstractions that make it easy for processes to communicate. Sleep and wakeup fill that void, allowing one process to sleep waiting for an event and another process to wake it up once the event has happened.

To illustrate what we mean, let's consider a simple producer/consumer queue. The queue allows one process to send a nonzero pointer to another process. Assuming there is only one sender and one receiver and they execute on different CPUs, this implementation is correct:

```
struct q {
  void *ptr;
};

void*
send(struct q *q, void *p)
{
  while(q->ptr != 0)
    ;
  q->ptr = p;
}

void*
recv(struct q *q)
{
  void *p;

  while((p = q->ptr) == 0)
    ;
  q->ptr = 0;
  return p;
}
```

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

The implementation above may be correct, but it is very expensive. If the sender sends rarely, the receiver will spend most of its time spinning in the `while` loop hoping for a pointer. The receiver's CPU could find more productive work if there were a way for the receiver to be notified when the `send` had delivered a pointer. `Sleep` and `wakeup` provide such a mechanism. `Sleep(chan)` sleeps on the pointer `chan`, called the wait channel, which may be any kind of pointer; it is used only as an identifying address and is not dereferenced. `Sleep` puts the calling process to sleep, releasing the CPU for other work. It does not return until the process is awake again. `Wakeup(chan)` wakes all the processes sleeping on `chan` (if any), causing their `sleep` calls to return. We can change the queue implementation to use `sleep` and `wakeup`:

```
void*
send(struct q *q, void *p)
{
  while(q->ptr != 0)
    ;
  q->ptr = p;
  wakeup(q);  /* wake recv */
}
```

```
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    }
```

This code is more efficient but no longer correct, because it suffers from what is known as the "lost wake up" problem. Suppose that `recv` finds that `q->ptr == 0` on line 215 and decides to call `sleep`. Before `recv` can sleep, `send` runs on another CPU: it changes `q->ptr` to be nonzero and calls `wakeup`, which finds no processes sleeping. Now `recv` continues executing at line 216: it calls `sleep` and goes to sleep. This causes a problem: `recv` is asleep waiting for a pointer that has already arrived. The next `send` will sleep waiting for `recv` to consume the pointer in the queue, at which point the system will be deadlocked.

The root of this problem is that the invariant that `recv` only sleeps when `q->ptr == 0` is violated by `send` running at just the wrong moment. To protect this invariant, we introduce a lock, which `sleep` releases only after the calling process is asleep; this avoids the missed wakeup in the example above. Once the calling process is awake again `sleep` reacquires the lock before returning. The following code is correct and makes efficient use of the CPU when `recv` must wait:

```
struct q {
  struct spinlock lock;
  void *ptr;
};

void*
send(struct q *q, void *p)
{
  lock(&q->lock);
  while(q->ptr != 0)
    ;
  q->ptr = p;
  wakeup(q);
  unlock(&q->lock);
}

void*
recv(struct q *q)
{
  void *p;

  lock(&q->lock);
  while((p = q->ptr) == 0)
    sleep(q, &q->lock);
  q->ptr = 0;
  unlock(&q->lock);
```

```
    return p;
}
```

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

Code: Sleep and wakeup

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

`Sleep` (2003) begins with a few sanity checks: there must be a current process (2005-2006) and `sleep` must have been passed a lock (2008-2009). Then `sleep` acquires `ptable.lock` (2018). 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` is done, 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 (2017).

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` (2023-2025).

At some point later, a process will call `wakeup(chan)`. `Wakeup` (2053) acquires `ptable.lock` and calls `wakeup1`, which does the real work. It is important that `wakeup` hold the `ptable.lock` both because it is manipulating process states and because, as we just saw, `ptable.lock` makes sure that `sleep` and `wakeup` do not miss each other. (`Wakeup1` is a separate function because sometimes the scheduler needs to execute a wakeup when it already holds the `ptable.lock`; we will see an example of this later.) `Wakeup1` (2053) loops over the process table. When it finds a process in state `SLEEPING` with a matching `chan`, it changes that process's state to `RUNNABLE`. The next time the scheduler runs, it will see that the process is ready to be run.

There is another complication: spurious wakeups.

Code: Pipes

The simple queue we used earlier in this Chapter was a toy, but xv6 contains a real queue that uses `sleep` and `wakeup` to synchronize readers and writers. That queue is the implementation of pipes. We saw the interface for pipes in Chapter 0: bytes written to one end of a pipe are copied in an in-kernel buffer and then can be read out of the other end of the pipe. Future chapters will examine the file system support surrounding pipes, but let's look now at the implementations of `pipewrite` and piperead.

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

Pipewrite (5230) begins by acquiring the pipe's lock, which protects the counts, the data, and their associated invariants. Piperead (5251) then tries to acquire the lock too, but cannot. It spins in acquire (1373) 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 (5244). During this loop, it could happen that the buffer fills (5236). In this case, pipewrite calls wakeup to alert any sleeping readers to the fact that there is data waiting in the buffer and then sleeps on &p->nwrite to wait for a reader to take some bytes out of the buffer. Sleep releases p->lock as part of putting pipewrite's process to sleep.

Now that p->lock is available, piperead manages to acquire it and start running in earnest: it finds that p->nread != p->nwrite (5256) (pipewrite went to sleep because p->nwrite == p->nread+PIPESIZE (5236)) so it falls through to the for loop, copies data out of the pipe (5263-5267), and increments nread by the number of bytes copied. That many bytes are now available for writing, so piperead calls wakeup (5268) to wake any sleeping writers before it returns to its caller.

Wakeup finds a process sleeping on &p->nwrite, the process that was running pipewrite but stopped when the buffer filled. It marks that process as RUNNABLE.

Let's suppose that the scheduler on the other CPU has decided to run some other process, so pipewrite does not start running again immediately. Instead, piperead returns to its caller, who then calls piperead again. Let's also suppose that the first piperead consumed all the data from the pipe buffer, so now p->nread == p->nwrite. Piperead sleeps on &p->nread to await more data (5261). Once the process calling piperead is asleep, the CPU can run pipewrite's process, causing sleep to return (5242). Pipewrite finishes its loop, copying the remainder of its data into the buffer (5244). Before returning, pipewrite calls wakeup in case there are any readers waiting for the new data (5246). There is one, the piperead we just left. It continues running (pipe.c/piperead-sleep/) and copies the new data out of the pipe.

Code: Wait and exit

Sleep and wakeup do not have to be used for implementing queues. They work for any condition that can be checked in code and needs to be waited for. As we saw in Chapter 0, a parent process can call wait to wait for a child to exit. In xv6, when a child exits, it does not die immediately. Instead, it switches to the ZOMBIE process state until the parent calls wait to learn of the exit. The parent is then responsible for freeing the memory associated with the process and preparing the struct proc for reuse.

Each process structure keeps a pointer to its parent in `p->parent`. If the parent exits before the child, the initial process `init` adopts the child and waits for it. This step is necessary to make sure that some process cleans up after the child when it exits. All the process structures are protected by `ptable.lock`.

`Wait` begins by acquiring `ptable.lock`. Then it scans the process table looking for children. If `wait` finds that the current process has children but that none of them have exited, it calls `sleep` to wait for one of the children to exit (2188) and loops. Here, the lock being released in `sleep` is `ptable.lock`, the special case we saw above.

`Exit` acquires `ptable.lock` and then wakes the current process's parent (2126). This may look premature, since `exit` has not marked the current process as a `ZOMBIE` yet, but it is safe: although the parent is now marked as `RUNNABLE`, the loop in `wait` cannot run until `exit` releases `ptable.lock` by calling `sched` to enter the scheduler, so `wait` can't look at the exiting process until after the state has been set to `ZOMBIE` (2138). Before exit reschedules, it reparents all of the exiting process's children, passing them to the `initproc` (2128-2135). Finally, `exit` calls `sched` to relinquish the CPU.

Now the scheduler can choose to run the exiting process's parent, which is asleep in `wait` (2188). The call to `sleep` returns holding `ptable.lock`; `wait` rescans the process table and finds the exited child with `state == ZOMBIE`. (2132). It records the child's `pid` and then cleans up the `struct proc`, freeing the memory associated with the process (2168-2175).

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`: when the child runs `exit`, its stack sits in the memory allocated as `p->kstack`. The stack can only be freed once the child process has called `swtch` (via `sched`) and moved off it. This reason is the main one that the scheduler procedure runs on its own stack, and that xv6 organizes `sched` and `scheduler` as co-routines. Xv6 couldn't invoke the procedure `scheduler` directly from the child, because that procedure would then be running on a stack that might be removed by the parent process calling `wait`.

Scheduling concerns

XXX spurious wakeups

XXX checking p->killed

XXX thundering herd

Real world

`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` disabled interrupts. This sufficed because Unix ran on a single-CPU system. Because xv6 runs on multiprocessors, it added an explicit lock to `sleep`. FreeBSD's `msleep` takes the same approach. Plan 9's `sleep` uses a callback function that runs with the scheduling lock held just before going to sleep; the function serves as a last minute check of the sleep condition, to avoid missed wakeups. The Linux kernel's `sleep` uses an explicit process queue instead of a wait channel; the queue has its own internal lock. (XXX Looking at the code that seems not to be enough; what's going on?)

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

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

Exercises:

Sleep has to check lk != &ptable.lock to avoid a deadlock (2017-2020). It could eliminate the special case by replacing

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

with

```
release(lk);
acquire(&ptable.lock);
 .P2
Doing this would break
sleep.
How?

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.

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

Additional reading:

cox and mullender, semaphores.

pike et al, sleep and wakeup

Chapter 6

Buffer cache

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

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

Code: Data structures

Disk hardware traditionally presents the data on the disk as a numbered sequence of 512-byte blocks called sectors: sector 0 is the first 512 bytes, sector 1 is the next, and so on. The disk drive and buffer cache coordinate the use of disk sectors with a data structure called a buffer, `struct buf` (3000). Each buffer represents the contents of one sector on a particular disk device. The `dev` and `sector` fields give the device and sector number and the `data` field is an in-memory copy of the disk sector. The `data` is often out of sync with the disk: it might have not yet been read in from disk, or it might have been updated but not yet written out. The `flags` track the relationship between memory and disk: the `B_VALID` flag means that `data` has been read in, and the B_DIRTY flag means that `data` needs to be written out. The `B_BUSY` flag is a lock bit; it indicates that some process is using the buffer and other processes must not.

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

Code: Disk driver

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

The kernel initializes the disk driver at boot time by calling ideinit (3351) from main (1232). Ideinit initializes idelock (3322) and then must prepare the hardware. In Chapter 3, xv6 disabled all hardware interrupts. Ideinit calls picenable and ioapicenable to enable the IDE_IRQ interrupt (3356-3357). The call to picenable enables the interrupt on a uniprocessor; ioapicenable enables the interrupt on a multiprocessor, but only on the last CPU (ncpu-1): on a two-processor system, CPU 1 handles disk interrupts.

Next, ideinit probes the disk hardware. It begins by calling idewait (3358) to wait for the disk to be able to accept commands. The disk hardware presents status bits on port 0x1f7, as we saw in chapter 1. Idewait (3332) polls the status bits until the busy bit (IDE_BSY) is clear and the ready bit (IDE_DRDY) is set.

Now that the disk controller is ready, ideinit can check how many disks are present. It assumes that disk 0 is present, because the boot loader and the kernel were both loaded from disk 0, but it must check for disk 1. It writes to port 0x1f6 to select disk 1 and then waits a while for the status bit to show that the disk is ready (3360-3367). If not, ideinit assumes the disk is absent.

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

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

Iderw (3454) adds the buffer b to the end of the queue (3467-3471). If the buffer is at the front of the queue, iderw must send it to the disk hardware by calling idestart (3424-3426); otherwise the buffer will be started once the buffers ahead of it are taken care of.

Idestart (3375) issues either a read or a write for the buffer's device and sector, according to the flags. If the operation is a write, idestart must supply the data now

(3389) and the interrupt will signal that the data has been written to disk. If the operation is a read, the interrupt will signal that the data is ready, and the handler will read it.

Having added the request to the queue and started it if necessary, `iderw` must wait for the result. As discussed above, polling does not make efficient use of the CPU. Instead, `iderw` sleeps, waiting for the interrupt handler to record in the buffer's flags that the operation is done (3479-3480). While this process is sleeping, xv6 will schedule other processes to keep the CPU busy.

Eventually, the disk will finish its operation and trigger an interrupt. As we saw in Chapter 3, `trap` will call `ideintr` to handle it (2624). `Ideintr` (3402) consults the first buffer in the queue to find out which operation was happening. If the buffer was being read and the disk controller has data waiting, `ideintr` reads the data into the buffer with `insl` (3415-3417). Now the buffer is ready: `ideintr` sets `B_VALID`, clears `B_DIRTY`, and wakes up any process sleeping on the buffer (3419-3422). Finally, `ideintr` must pass the next waiting buffer to the disk (3424-3426).

Code: Interrupts and locks

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

Interrupts can cause concurrency even on a single processor: if interrupts are enabled, kernel code can be stopped at any moment to run an interrupt handler instead. Suppose `iderw` held the `idelock` and then got interrupted to run `ideintr`. `Ideintr` would try to lock `idelock`, see it was held, and wait for it to be released. In this situation, `idelock` will never be released—only `iderw` can release it, and `iderw` will not continue running until `ideintr` returns—so the processor, and eventually the whole system, will deadlock. To avoid this situation, if a lock is used by an interrupt handler, a processor must never hold that lock with interrupts enabled. Xv6 is more conservative: it never holds any lock with interrupts enabled. It uses `pushcli` (1455) and `popcli` (1466) to manage a stack of "disable interrupts" operations (`cli` is the x86 instruction that disables interrupts, as we saw in Chapter 1). `Acquire` calls `pushcli` before trying to acquire a lock (1375), and `release` calls `popcli` after releasing the lock (1421). It is important that `acquire` call `pushcli` before the `xchg` that might acquire the lock (1382). 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 (1382). These races are similar to the ones involving `holding` (see Chapter 4).

Code: Buffer cache

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

The buffer cache is a doubly-linked list of buffers. Binit, called by main (1229), initializes the list with the NBUF buffers in the static array buf (3550-3559). All other access to the buffer cache refer to the linked list via bcache.head, not the buf array.

Bread (3602) calls bget to get a locked buffer for the given sector (3606). If the buffer needs to be read from disk, bread calls iderw to do that before returning the buffer.

Bget (3566) scans the buffer list for a buffer with the given device and sector numbers (3573-3584). If there is such a buffer, bget needs to lock it before returning. If the buffer is not in use, bget can set the B_BUSY flag and return (3576-3583). If the buffer is already in use, bget sleeps on the buffer to wait for its release. When sleep returns, bget cannot assume that the buffer is now available. In fact, since sleep released and reacquired buf_table_lock, there is no guarantee that b is still the right buffer: maybe it has been reused for a different disk sector. Bget has no choice but to start over (3582), hoping that the outcome will be different this time.

If there is no buffer for the given sector, bget must make one, possibly reusing a buffer that held a different sector. It scans the buffer list a second time, looking for a block that is not busy: any such block can be used (3586-3588). Bget edits the block metadata to record the new device and sector number and mark the block busy before returning the block (3591-3593). Note that the assignment to flags not only sets the B_BUSY bit but also clears the B_VALID and B_DIRTY bits, making sure that bread will refresh the buffer data from disk rather than use the previous block's contents.

Because the buffer cache is used for synchronization, it is important that there is only ever one buffer for a particular disk sector. The assignments (3589-3591) are only safe because bget's first loop determined that no buffer already existed for that sector, and bget has not given up buf_table_lock since then.

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

Once bread has returned a buffer to its caller, the caller has exclusive use of the buffer and can read or write the data bytes. If the caller does write to the data, it must call bwrite to flush the changed data out to disk before releasing the buffer. Bwrite (3614) 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 (3624) moves the buffer from its position in the linked list to the front of the list (3631-3636), clears the B_BUSY bit, and wakes any processes sleeping on the buffer. Moving the buffer has the effect that the buffers are ordered by how recently they were used (meaning released): the first buffer in the list is the most recently used, and the last is the least recently used. The two loops in bget take advantage of this: the scan for an existing buffer must process the entire list in the worst case, but checking the most recently used buffers first (starting at bcache.head and following next pointers) will reduce scan time when there is good locality of reference. The scan to pick a buffer to reuse picks the least recently used block by scanning backward (following prev pointers); the implicit assumption is that the least recently used buffer is the one least likely to be used again soon.

Real world

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

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

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

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

Exercises

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

Chapter 7

File system data structures

The disk driver and buffer cache (Chapter 6) provide safe, synchronized access to disk blocks. Individual blocks are still a very low-level interface, too raw for most programs. Xv6, following Unix, provides a hierarchical file system that allows programs to treat storage as a tree of named files, each containing a variable length sequence of bytes. The file system is implemented in four steps. The first step is the block allocator. It manages disk blocks, keeping track of which blocks are in use, just as the memory allocator in Chapter 2 tracks which memory pages are in use. The second step is unnamed files called inodes (pronounced i-node). Inodes are a collection of allocated blocks holding a variable length sequence of bytes. The third step is directories. A directory is a special kind of inode whose content is a sequence of directory entries, each of which lists a name and a pointer to another inode. The last step is hierarchical path names like `/usr/rtm/xv6/fs.c`, a convenient syntax for identifying particular files or directories.

File system layout

Xv6 lays out its file system as follows. Block 0 is unused, left available for use by the operating system boot sequence. Block 1 is called the superblock; it contains metadata about the file system. After block 1 comes a sequence of inodes blocks, each containing inode headers. After those come bitmap blocks tracking which data blocks are in use, and then the data blocks themselves.

The header `fs.h` (3150) contains constants and data structures describing the layout of the file system. The superblock contains three numbers: the file system size in blocks, the number of data blocks, and the number of inodes.

Code: Block allocator

The block allocator is made up of the two functions: `balloc` allocates a new disk block and `bfree` frees one. `Balloc` (3704) starts by calling `readsb` to read the superblock. (`Readsb` (3678) is almost trivial: it reads the block, copies the contents into `sb`, and releases the block.) Now that `balloc` knows the number of inodes in the file system, it can consult the in-use bitmaps to find a free data block. The loop (3712) considers every block, starting at block 0 up to `sb.size`, the number of blocks in the file system, checking for a block whose bitmap bit is zero, indicating 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 inner loop checks all the bits in a single bitmap block—there are BPB—and the outer loop considers all the blocks in increments of BPB. There may be multiple processes calling balloc simultaneously, and yet there is no explicit locking. Instead, balloc relies on the fact that the buffer cache (bread and brelse) only let one process use a buffer at a time. When reading and writing a bitmap block (3714-3722), balloc can be sure that it is the only process in the system using that block.

Bfree (3730) is the opposite of balloc and has an easier job: there is no search. It finds the right bitmap block, clears the right bit, and is done. Again the exclusive use implied by bread and brelse avoids the need for explicit locking.

When blocks are loaded in memory, they are referred to by pointers to buf structures; as we saw in the last chapter, a more permanent reference is the block's address on disk, its block number.

Inodes

In Unix technical jargon, the term inode refers to an unnamed file in the file system, but the precise meaning can be one of three, depending on context. First, there is the on-disk data structure, which contains metadata about the inode, like its size and the list of blocks storing its data. Second, there is the in-kernel data structure, which contains a copy of the on-disk structure but adds extra metadata needed within the kernel. Third, there is the concept of an inode as the whole unnamed file, including not just the header but also its content, the sequence of bytes in the data blocks. Using the one word to mean all three related ideas can be confusing at first but should become natural.

Inode metadata is stored in an inode structure, and all the inode structures for the file system are packed into a separate section of disk called the inode blocks. Every inode structure is the same size, so it is easy, given a number n, to find the nth inode structure 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 structure is a struct dinode (3172). The type field in the inode header doubles as an allocation bit: a type of zero means the inode is available for use. The kernel keeps the set of active inodes in memory; its struct inode is the in-memory copy of a struct dinode on disk. The access rules for in-memory inodes are similar to the rules for buffers in the buffer cache: there is an inode cache, iget fetches an inode from the cache, and iput releases an inode. Unlike in the buffer cache, iget returns an unlocked inode: it is the caller's responsibility to lock the inode with ilock before reading or writing metadata or content and then to unlock the inode with iunlock before calling iput. Leaving locking to the caller allows the file system calls (described in Chapter 8) to manage the atomicity of complex operations. Multiple processes can hold a reference to an inode ip returned by iget (ip->ref counts exactly how many), but only one process can lock the inode at a time.

The inode cache is not a true cache: its only purpose is to synchronize access by multiple processes to shared inodes. It does not actually cache inodes when they are not being used; instead it assumes that the buffer cache is doing a good job of avoiding unnecessary disk acceses and makes no effort to avoid calls to bread. The in-memory copy of the inode augments the disk fields with the device and inode number, the reference count mentioned earlier, and a set of flags.

Code: Inodes

To allocate a new inode (for example, when creating a file), xv6 calls `ialloc` (3802). `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` (3818). Like in `balloc`, the correct operation of `ialloc` depends on the fact that only one process at a time can be holding a reference to `bp`: `ialloc` can be sure that some other process does not simultaneously see that the inode is available and try to claim it.

`Iget` (3853) 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. (3862-3866). As `iget` scans, it records the position of the first empty slot (3867-3868), which it uses if it needs to allocate a new cache entry. In both cases, `iget` returns one reference to the caller: it is the caller's responsibility to call `iput` to release the inode. It can be convenient for some callers to arrange to call `iput` multiple times. `Idup` (3888) increments the reference count so that an additional `iput` call is required before the inode can be dropped from the cache.

Callers must lock the inode using `ilock` before reading or writing its metadata or content. `Ilock` (3902) uses a now-familiar sleep loop to wait for `ip->flag`'s `I_BUSY` bit to be clear and then sets it (3911-3913). Once `ilock` has exclusive access to the inode, it can load the inode metadata from the disk (more likely, the buffer cache) if needed. `Iunlock` (3934) clears the `I_BUSY` bit and wakes any processes sleeping in `ilock`.

`Iput` (3952) releases a reference to an inode by decrementing the reference count (3968). If this is the last reference, so that the count would become zero, the inode is about to become unreachable: its disk data needs to be reclaimed. `Iput` relocks the inode; calls `itrunc` to truncate the file to zero bytes, freeing the data blocks; sets the type to 0 (unallocated); writes the change to disk; and finally unlocks the inode (3955-3967).

The locking protocol in `iput` deserves a closer look. The first part with examining is that when locking `ip`, `iput` simply assumed that it would be unlocked, instead of using a sleep loop. This must be the case, because the caller is required to unlock `ip` before calling `iput`, and the caller has the only reference to it (`ip->ref == 1`). The second part worth examining is that `iput` temporarily releases (3960) and reacquires (3964) the cache lock. This is necessary because `itrunc` and `iupdate` will sleep during disk i/o, but we must consider what might happen while the lock is not held. Specifically, once `iupdate` finishes, the on-disk structure is marked as available for use, and a concurrent call to `ialloc` might find it and reallocate it before `iput` can finish. `Ialloc` will return a reference to the block by calling `iget`, which will find `ip` in the cache, see that its `I_BUSY` flag is set, and sleep. Now the in-core inode is out of sync compared to the disk: `ialloc` reinitialized the disk version but relies on the caller to load it into memory during `ilock`. In order to make sure that this happens, `iput` must clear not only I_BUSY but also I_VALID before releasing the inode lock. It does this by zeroing flags (3965).

Code: Inode contents

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

Bmap (4010) begins by picking off the easy case: the first NDIRECT blocks are listed in the inode itself (4015-4019). The next NINDIRECT blocks are litsed in the indirect block at ip->addrs[NDIRECT]. Bmap reads the indirect block (4026) and then reads a block number from the right position within the block (4027). If the block number exceeds NDIRECT+NINDIRECT, bmap panics: callers are responsible for not asking about out-of-range block numbers.

Bmap allocates block as needed. Unallocated blocks are denoted by a block number of zero. As bmap encouters zeros, it replaces them with the numbers of fresh blocks, allocated on demand. (4016-4017, 4024-4025).

Bmap allocates blocks on demand as the inode grows; itrunc frees them, resetting the inode's size to zero. Itrunc (4054) starts by freeing the direct blocks (4060-4065) and then the ones listed in the indirect block (4070-4073), and finally the indirect block itself (4075-4076).

Bmap makes it easy to write functions to access the inode's data stream, like readi and writei. Readi (4102) reads data from the inode. It starts making sure that the offset and count are not reading beyond the end of the file. Reads that start beyond the end of the file return an error (4113-4114) while reads that start at or cross the end of the file return fewer bytes than requested (4115-4116). The main loop processes each block of the file, copying data from the buffer into dst (4118-4123). Writei (4152) 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 (4165-4166); the loop copies data into the buffers instead of out (4171); and if the write has extended the file, writei must update its size (4176-4179).

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 Chapter 8.

Stati (3674) copies inode metadata into the stat structure, which is exposed to user programs via the stat system call (see Chapter 8).

Code: Directories

Xv6 implements a directory as a special kind of file: it has type `T_DEV` and its data is a sequence of directory entries. Each entry is a `struct dirent` (3203), which contains a name and an inode number. The name is at most `DIRSIZ` (14) letters; if shorter, it is terminated by a NUL (0) byte. Directory entries with inode number zero are unallocated.

`Dirlookup` (4212) searches the directory for an entry with the given name. If it finds one, it returns 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. The outer for loop (4221) considers each block in the directory in turn; the inner loop (4223) considers each directory entry in the block, ignoring entries with inode number zero. If `dirlookup` finds an entry with the right name, it updates `*poff`, releases the block, and returns an unlocked inode obtained via `iget` (4228-4235). `Dirlookup` is the reason that `iget` returns unlocked inodes. The caller has locked `dp`, so if the lookup was for `.`, an alias for the current directory, attempting to lock the inode before returning would try to re-lock `dp` and deadlock. (There are more complicated deadlock scenarios involving multiple processes and `..`, an alias for the parent directory; `.` is not the only problem.) The caller can unlock `dp` and then lock `ip`, ensuring that it only holds one lock at a time.

If `dirlookup` is read, `dirlink` is write. `Dirlink` (4252) 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 (4258-4262). The main loop reads directory entries looking for an unallocated entry. When it finds one, it stops the loop early (4268-4269), with `off` set to the offset of the available entry. Otherwise, the loop ends with `off` set to `dp->size`. Either way, `dirlink` then adds a new entry to the directory by writing at offset `off` (4272-4275).

`Dirlookup` and `dirlink` use different loops to scan the directory: `dirlookup` operates a block at a time, like `balloc` and `ialloc`, while `dirlink` operates one entry at a time by calling `readi`. The latter approach calls `bread` more often—once per entry instead of once per block—but is simpler and makes it easy to exit the loop with `off` set correctly. The more complex loop in `dirlookup` does not save any disk i/o—the buffer cache avoids redundant reads—but doe savoid repeated locking and unlocking of `bcache.lock` in `bread`. The extra work may have been deemed necessary in `dirlooup` but not `dirlink` because the former is so much more common than the latter. (TODO: Make this paragraph into an exercise?)

Path names

The code examined so far implements a hierarchical file system. The earliest Unix systems, such as the version described in Thompson and Ritchie's earliest paper, stops here. Those systems looked up names in the current directory only; to look in another directory, a process needed to first move into that directory. Before long, it became clear that it would be useufl to refer to directories further away: the name `xv6/fs.c` means first look up `xv6`, which must be a directory, and then look up `fs.c` in that directory. A path beginning with a slash is called rooted. The name `/xv6/fs.c` is like `xv6/fs.c` but starts the lookup at the root of the file system tree instead of the current directory. Now, decades later, hierarchical, optionally rooted path names are so commonplace that it is easy to forgoet that they had to be invented; Unix did that. (TODO: Is this really true?)

Code: Path names

The final section of `fs.c` interprets hierarchical path names. `Skipelem` (4315) helps parse them. It copies the first element of `path` to `name` and retrns a pointer to the remainder of `path`, skipping over leading slashes. Appendix A examines the implementation in detail.

`Namei` (4389) evaluates `path` as a hierarchical path name and returns the corresponding `inode`. `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` (4354) starts by deciding where the path evaluation begins. If the path begins with a slash, evaluation begins at the root; otherwise, the current directory (4358-4361). Then it uses `skipelem` to consider each element of the path in turn (4363). 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 (4364-4368). (Locking `ip` is necessary not because `ip->type` can change underfoot—it can't—but because until `ilock` runs, `ip->type` is not guaranteed to have been loaded from disk.) If the call is `nameiparent` and this is the last path element, the loop stops early, as per the definition of `nameiparent`; the final path element has already been copied into `name`, so `namex` need only return the unlocked `ip` (4369-4373). Finally, the loop looks for the path element using `dirlookup` and prepares for the next iteration by setting `ip.=.next` (4374-4379). When the loop runs out of path elements, it returns `ip`.

TODO: It is possible that namei belongs with all its uses, like open and close, and not here in data structure land.

Real world

Xv6's file system implementation assumes that disk operations are far more expensive than computation. It uses an efficient tree structure on disk but comparatively inefficient linear scans in the inode and buffer cache. The caches are small enough and disk accesses expensive enough to justify this tradeoff. Modern operating systems with larger caches and faster disks use more efficient in-memory data structures. The disk structure, however, with its inodes and direct blocks and indirect blocks, has been remarkably persistent. 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, especially if the entries in each disk block can be kept sorted, but when directories span many disk blocks. Microsoft Windows's NTFS, Mac OS X's HFS, and Solaris's ZFS, just to name a few, implement a directory as an on-disk balanced tree of blocks. This is more complicated than reusing the file implementation but guarantees logarithmic-time directory lookups.

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

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

Other features: snapshotting and backup.

Exercises

Exercise: why panic in balloc? Can we recover?

Exercise: why panic in ialloc? Can we recover?

Exercise: inode generation numbers.

Chapter 8

File system calls

The previous chapter layed described the file system data structures, and how they are used to implemented files and directories. This chapter completes the file system by explaning how the system calls for file operations are implemented. In this chapter, "file" means an open file.

Code: Files

Xv6 gives each process its own table of open files, as we saw in Chapter 0. Each open file is represented by a `struct file` (3250), which is a wrapper around either an inode or a pipe, plus an i/o offset. Each call to `open` creates a new open file (a new `struct file`): if multiple processes open the same file independently, the different instances will have different i/o offsets. On the other hand, a single open file (the same `struct file`) can appear multiple times in one process's file table and also in the file tables of multiple processes. This would happen if one process used `open` to open the file and then created aliases using `dup` or shared it with a child using `fork`. A reference count tracks the number of references to a particular open file. A file can be open for reading or writing or both. The `readable` and `writable` fields track this.

All the open files in the system are kept in a global file table, the `ftable`. Like the inode cache, 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` (4421) scans the file table for an unreferenced file (`f->ref == 0`) and returns a new reference; `filedup` (4439) increments the reference count; and `fileclose` (4452) decrements it. When a file's reference count reaches zero, `fileclose` releases the underlying pipe or inode, according to the type.

`Filestat`, `fileread`, and `filewrite` implement the `stat`, `read`, and `write` operations on files. `Filestat` (4476) is only allowed on inodes and calls `stati`. `Fileread` and `filewrite` check that the operation is allowed by the open mode and then pass the call through to either the pipe or inode implementation. If the file represents an inode, `fileread` and `filewrite` use the i/o offset as the offset for the operation and then advance it (4512-4513, 4532-4533). Pipes have no concept of offset. Remember from Chapter 7 that the inode functions require the caller to handle locking (4479-4481, 4511-4514, 4531-4534). The inode locking has the convenient side effect that the read and write offsets are updated atomically, so that multiple writing to the same file simultaneously cannot overwrite each other's data, though their writes may end up interlaced.

Code: System calls

Chapter 3 introduced helper functions for implementing system calls: `argint`, `argstr`, and `argptr`. The file system adds another: `argfd` (4563) interprets the `n`th argument as a file descriptor. It calls `argint` to fetch the integer `fd` and then checks that `fd` is a valid file table index. Although `argfd` returns a reference to the file in `*pf`, it does not increment the reference count: the caller shares the reference from the file table. As we will see, this convention avoids reference count operations in most system calls.

The function `fdalloc` (4582) helps manage the current process's file table: it scans the table for an open slot, and if it finds one, inserts `f` and returns the index of the slot, which will serve as the file descriptor. It is up to the caller to manage the reference count.

Finally we are ready to implement system calls. The simplest is `sys_dup` (4601), which makes use of both of these helpers. It calls `argfd` to obtain the file corresponding to the system call argument and then calls `fdalloc` to assign it an additional file descriptor. If both are successful, it calls `filedup` to adjust the reference count: `fdalloc` has created a new reference. Similarly, `sys_close` (4639) obtains a file, removes it from the file table, and releases the reference.

`Sys_read` (4615) parses its arguments as a file descriptor, a pointer, and a size and then calls `fileread`. Note that no reference count operations are necessary: `sys_read` is piggybacking on the reference in the file table. The reference cannot disappear during the `sys_read` because each process has its own file table, and it is impossible for the process to call `sys_close` while it is in the middle of `sys_read`. `Sys_write` (4627) is identical to `sys_read` except that it calls `filewrite`. `Sys_fstat` (4651) is very similar to the previous two.

`Sys_link` and `sys_unlink` edit directories, creating or removing references to inodes. They are another good example of the power of exposing the file system locking to higher-level functions.

`Sys_link` (4663) begins by fetching its arguments, two strings `old` and `new` (4668). Assuming `old` exists and is not a directory (4670-4676), `sys_link` increments its `ip->nlink` count—the number of directories in which it appears—and flushes the new count to disk (4677-4678). Then `sys_link` calls `nameiparent` to find the parent directory and final path element of `new` (4681) and creates a new directory entry pointing at `old`'s inode (4684). 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`.

`Sys_link` would have simpler control flow and error handling if it delayed the increment of `ip->nlink` until it had successfully created the link, but doing this would put the file system temporarily in an unsafe state. The low-level file system code in Chapter 7 was careful not to write out pointers to disk blocks before writing the disk blocks themselves, lest the machine crash with a file system with pointers to old blocks. The same principle is being used here: to avoid dangling pointers, it is important that the link count always be at least as large as the true number of links. If the system crashed after sys_link creating the second link but before it incremented ip->nlink, then the file system would have an inode with two links but a link count set to one. Removing one of the links would cause the inode to be reused even though there was still a reference to it.

Sys_unlink (4751) is the opposite of sys_link: it removes the named path from the file system. It calls nameiparent to find the parent directory, sysfile.c:/nameiparent.path/, checks that the final element, name, exists in the directory (4770), clears the directory entry (4785), and then updates the link count (4793). As was the case for sys_link, the order here is important: sys_unlink must update the link count only after the directory entry has been removed. There are a few more steps if the entry being removed is a directory: it must be empty (4778) and after it has been removed, the parent directory's link count must be decremented, to reflect that the child's .. entry is gone.

Sys_link creates a new name for an existing inode. Create (4801) creates a new name for a new inode. It is a generalization of the three file creation system calls: open with the O_CREATE flag makes a new ordinary file, mkdir makes a new directory , and mkdev makes a new device file. Like sys_link, create starts by caling nameiparent to get the inode of the parent directory. It then calls dirlookup to check whether the name already exists (4811). 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 (4815). Otherwise, it is an error (4816-4817). If the name does not already exist, create now allocates a new inode with ialloc (4820). 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 (4833). 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 (4851) is the most complex, because creating a new file is only a small part of what it can do. If open is passed the O_CREATE flag, it calls create (4862). Otherwise, it calls namei (4865). 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 (4874) and then fills in the file (4882-4886). Since we have been so careful to initialize data structures before creating pointers to them, this sequence should feel wrong, but it is safe: no other process can access the partially initialized file since it is only in the current process's table, and these data structures are in memory, not on disk, so they don't persist across a machine crash.

Sys_mkdir (4901) and sys_mknod (4913) are trivial: they parse their arguments, call create, and release the inode it returns.

Sys_chdir (4930) changes the current directory, which is stored as cp->cwd rather than in the file table. It evaluates the new path, checks that it is a directory, releases the old `cp->cwd`, and saves the new one in its place.

Chapter 5 examined the implementation of pipes before we even had a file system. `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. Chapter 5 did not examine `pipealloc` (5171) and `pipeclose` (5211), but they should be straightforward after walking through the examples above.

The final file system call is `exec`, which is the topic of the next chapter.

Real world

The file system interface in this chapter has proved remarkably durable: modern systems such as BSD and Linux continue to be based on the same core system calls. In those systems, multiple processes (sometimes called threads) can share a file descriptor table. That introduces another level of locking and complicates the reference counting here.

Xv6 has two different file implementations: pipes and inodes. Modern Unix systems have many: pipes, network connections, and inodes from many different types of file systems, including network file systems. Instead of the `if` statements in `fileread` and `filewrite`, these systems typically give each open file a table of function pointers, one per operation, and call the function pointer to invoke that inode's implementation of the call. Network file systems and user-level file systems provide functions that turn those calls into network RPCs and wait for the response before returning. Network file systems are now an everyday occurrence, but networking in general is beyond the scope of this book. On the other hand, the World Wide Web is in some ways a global-scale hierarchical file system.

Exercises

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

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

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

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

Chapter 9

Exec

Chapter 2 stopped with the `initproc` invoking the kernel's `exec` system call. As a result, we took detours into interrupts, multiprocessing, device drivers, and a file system. With these taken care of, we can finally look at the implementation of `exec`. As we saw in Chapter 0, `exec` replaces the memory and registers of the current process with a new program, but it leaves the file descriptors, process id, and parent process the same. `Exec` is thus little more than a binary loader, just like the one in the boot sector from Chapter 1. The additional complexity comes from setting up the stack. The memory image of an executing process looks like:

```
[XXX better picture: not ASCII art,
show individual argv pointers, show argc,
show argument strings, show fake return address.]

+--------------------------------------+
| text | data | stack | args | heap ... |
+--------------------------------------+
                      ^
                      |
                  initial sp
```

In xv6, the stack is a single page—4096 bytes—long. The command-line arguments follow the stack immediately in memory, so that the program can start at `main` as if the function call `main(argc, argv)` had just started. The heap comes last so that expanding it does not require moving any of the other sections.

Code

When the system call arrives, `syscall` invokes `sys_exec` via the `syscalls` table (2879). `Sys_exec` (4951) parses the system call arguments, as we saw in Chapter 3, and invokes `exec` (4972).

`Exec` (5009) opens the named binary `path` using `namei` (5021) and then reads the ELF header. Like the boot sector, it uses `elf.magic` to decide whether the binary is an ELF binary (5025-5029). Then it makes two passes through the program segment and argument lists. The first computes the total amount of memory needed, and the second creates the memory image. The total memory size includes the program segments (5032-5041), the argument strings (5043-5048), the argument vector pointed at by `argv` (5049), the `argv` and `argc` arguments to `main` (5049-5051), and the stack (5053-5054). `Exec` then allocates and zeros the required amount of memory (5056-5061) and copies the data into the new memory image: the program segments (5063-5077), the argument strings and pointers (5079-5090), and the stack frame for `main` (5092-5099).

Notice that when `exec` copies the program segments, it makes sure that the data being loaded into memory fits in the declared size `ph.memsz` (5069-5070). Without this check, a malformed ELF binary could cause `exec` to write past the end of the allocated memory image, causing memory corruption and making the operating system unstable. The boot sector neglected this check both to reduce code size and because not checking doesn't change the failure mode: either way the machine doesn't boot if given a bad ELF image. In contrast, in `exec` this check is the difference between making one process fail and making the entire system fail.

During the preparation of the new memory image, if `exec` detected an error like an invalid program segment, it jumps to the label `bad`, frees the new image, and returns −1. `Exec` must wait to free the old image until it is sure that the system call will succeed: if the old image is gone, the system call cannot return −1 to it. The only error cases in `exec` happen during the creation of the image. Once the image is complete, `exec` can free the old image and install the new one (5107-5111). After changing the image, `exec` must update the user segment registers to refer to the new image, just as `sbrk` did (5112). Finally, `exec` returns 0. Success!

Now the `initcode` (6700) is done. `Exec` has replaced it with the real `/init` binary, loaded out of the file system. `Init` (6810) 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

`Exec` is the most complicated code in xv6 in and in most operating systems. It involves pointer translation (in `sys_exec` too), many error cases, and must replace one running process with another. Real world operationg systems have even more complicated `exec` implementations. They handle shell scripts (see exercise below), more complicated ELF binaries, and even multiple binary formats.

Exercises

1. 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.