DRAFT as of September 12, 2009: Copyright 2009 Cox, Kaashoek, Morris

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 alter-

nates 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.