Chapter 0 Operating system interfaces

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

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

This book uses a single operating system as a concrete example to illustrate operating system concepts. That operating system, xv6, provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie's Unix operating system, as well as mimicking Unix's internal design. Unix provides a narrow interface whose mechanismscombine well, offering a surprising degree of generality. This interface has been sosuccessful that modern operating systems—BSD, Linux, Mac OS X, Solaris, and even,to a lesser extent, Microsoft Windows—have Unix-like interfaces. Understanding xv6is a good start toward understanding any of these systems and many others.

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

When a process needs to invoke a kernel service, it invokes a procedure call inthe operating system interface. Such procedures are call system calls. The systemcall 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 eachprocess executing in user space can access only its own memory. The kernel executeswith the hardware privileges required to implement these protections; user programsexecute without those privileges. When a user program invokes a system call, thehardware raises the privilege level and starts executing a prearranged function in thekernel.

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 Unixkernels traditionally offer. The calls are:

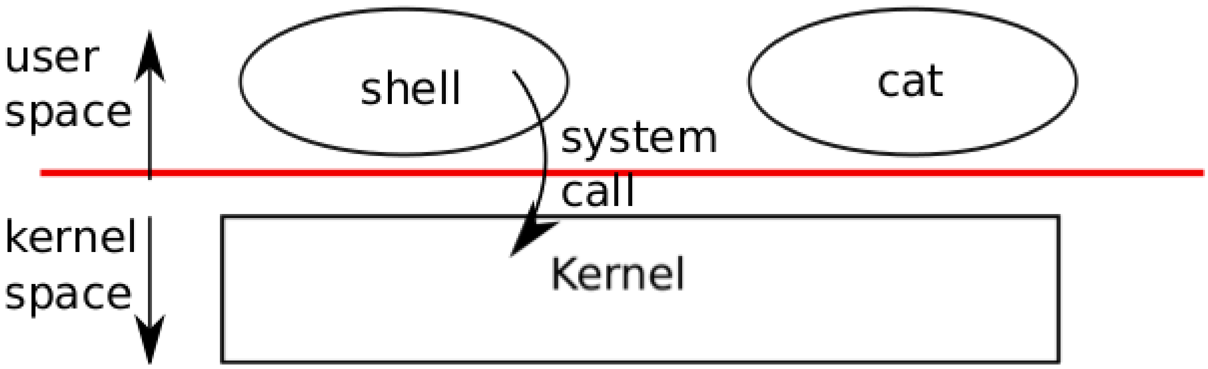

Figure 0-1. A kernel and two user processes.

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

System call	Description
unlink(filename)	Remove a file

The rest of this chapter outlines xv6's services—processes, memory, file descriptors, pipes, and file system—and illustrates them with code snippets and discussions ofhow the shell uses them.

The shell's use of system calls illustrates how carefully theyhave been designed.The shell is an ordinary program that reads commands from the user and executes them, and is the primary user interface to traditional Unix-like systems. The factthat the shell is a user program, not part of the kernel, illustrates the power of the system call interface: there is nothing special about the shell. It also means that the shellis easy to replace; as a result, modern Unix systems have a variety of shells to choosefrom, each with its own user interface and scripting features. The xv6 shell is a simpleimplementation of the essence of the Unix Bourne shell. Its implementation can befound at line (7850).

Processes and memory

An xv6 process consists of user-space memory (instructions, data, and stack) andperprocess state private to the kernel. Xv6 provides timesharing: it transparentlyswitches the available CPUs among the set of processes waiting to execute. When aprocess is not executing, xv6 saves its CPU registers, restoring them when it next runsthe process. The kernel associates a process identifier, or pid, with each process.

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

```
int pid;



pid = fork();

if(pid > 0){

  printf("parent: child=%d\n", pid);

  pid = wait();

  printf("child %d is done\n", pid);

} else if(pid == 0){

  printf("child: exiting\n");

  exit();

} else {

  printf("fork error\n");
```

```
}
```

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

```
parent: child=1234

child: exiting
```

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

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

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

The exec system call replaces the calling process's memory with a new memoryimage loaded from a file stored in the file system. The file must have a particular format, which specifies which part of the file holds instructions, which part is data, atwhich instruction to start, etc. xv6 uses the ELF format, which Chapter 2 discusses inmore 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 theELF header. Exec takes two arguments: the name of the file containing the executableand 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 firstargument, which is conventionally the name of the program.

The xv6 shell uses the above calls to run programs on behalf of users. The mainstructure of the shell is simple; see main (8001). The main loop reads the input on thecommand line using getcmd. Then it calls fork, which creates a copy of the shell process. The parent shell calls wait, while the child process runs the command. For example, if the user had typed ‘‘echo hello’’ at the prompt, runcmd would have beencalled with ‘‘echo hello’’ as the argument. runcmd (7906) runs the actual command.For ‘‘echo hello’’, it would call exec (7926). If exec succeeds then the child will execute instructions from echo instead of runcmd. At some point echo will call exit,which will cause the parent to return from wait in main (8001). You might wonderwhy fork and exec are not combined in a single call; we will see later that separatecalls for creating a process and loading a program is a clever design.

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

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

I/O and File descriptors

A file descriptor is a small integer representing a kernel-managed object thata process may read from or write to. A process may obtain a file descriptor by opening a file, directory, or device, or by creating a pipe, or by duplicating an existing descriptor. For simplicity we'll often refer to the object a file descriptor refers to as a''file''; the file descriptor interface abstracts away the differences between files, pipes,and devices, making them all look like streams of bytes.

Internally, the xv6 kernel uses the file descriptor as an index into a perprocess table, so that every process has a private space of file descriptors starting at zero. Byconvention, a process reads from file descriptor 0 (standard input), writes output to filedescriptor 1 (standard output), and writes error messages to file descriptor 2 (standarderror). As we will see, the shell exploits the convention to implement I/O redirectionand pipelines. The shell ensures that it always has three file descriptors open (8007),which are by default file descriptors for the console.

The read and write system calls read bytes from and write bytes to open filesnamed by file descriptors. The call read(fd, buf, n) reads at most n bytes from the file descriptor fd, copies them into buf, and returns the number of bytes read. Eachfile descriptor that refers to a file has an offset associated with it. Read reads datafrom the current file offset and then advances that offset by the number of bytes read:a subsequent read will return the bytes following the ones returned by the first read.When there are no more bytes to read, read returns zero to signal the end of the file.

The call write(fd, buf, n) writes n bytes from buf to the file descriptor fd andreturns 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 advancesthat offset by the number of bytes written: each write picks up where the previousone left off.

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

```
char buf[512];

int n;

for(;;){

  n = read(0, buf, sizeof buf);

  if(n == 0)

    break;

  if(n < 0){

    fprintf(2, "read error\n");

    exit();
```

```
  }

  if(write(1, buf, n) != n){

    fprintf(2, "write error\n");

    exit();

  }

}
```

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

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

File descriptors and fork interact to make I/O redirection easy to implement.Fork copies the parent’s file descriptor table along with its memory, so that the childstarts with exactly the same open files as the parent. The system call exec replaces thecalling process’s memory but preserves its file table. This behavior allows the shell toimplement 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 thecommand 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 forthe newly opened input.txt: 0 will be the smallest available file descriptor. Cat thenexecutes with file descriptor 0 (standard input) referring to input.txt.

The code for I/O redirection in the xv6 shell works in exactly this way (7930). Recall that at this point in the code the shell has already forked the child shell and thatruncmd will call exec to load the new program. Now it should be clear why it is agood idea that fork and exec are separate calls. This separation allows the shell to fixup the child process before the child runs the intended program.

Although fork copies the file descriptor table, each underlying file offset is sharedbetween 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 datahello world. The write in the parent (which, thanks to wait, runs only after thechild is done) picks up where the child's write left off. This behavior helps producesequential output from sequences of shell commands, like (echo hello; echo world)>output.txt.

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

```
fd = dup(1);

write(1, "hello ", 6);

write(fd, "world\n", 6);
```

Two file descriptors share an offset if they were derived from the same originalfile descriptor by a sequence of fork and dup calls. Otherwise file descriptors do notshare offsets, even if they resulted from open calls for the same file. Dup allows shellsto implement commands like this: ls existing-file non-existing-file > tmp12>&1. The 2>&1 tells the shell to give the command a file descriptor 2 that is a duplicate of descriptor 1. Both the name of the existing file and the error message for thenon-existing file will show up in the file tmp1. The xv6 shell doesn't support I/O redirection for the error file descriptor, but now you know how to implement it.

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

Pipes

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

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

```
int p[2];

char *argv[2];

argv[0] = "wc";

argv[1] = 0;

pipe(p);

if(fork() == 0) {

  close(0);

  dup(p[0]);

  close(p[0]);

  close(p[1]);

  exec("/bin/wc", argv);

} else {

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

  close(p[0]);

  close(p[1]);

}
```

The program calls pipe, which creates a new pipe and records the read and write filedescriptors 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 thepipe. The parent writes to the write end of the pipe and then closes both of its filedescriptors.

If no data is available, a read on a pipe waits for either data to be written or allfile 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 blocksuntil it is impossible for new data to arrive is one reason that it’s important for thechild to close the write end of the pipe before executing wc above: if one of wc’s filedescriptors referred to the write end of the pipe, wc would never see end-of-file.

The xv6 shell implements pipelines such as grep fork sh.c | wc -l in a manner similar to the above code (7950). The child process creates a pipe to connect theleft end of the pipeline with the right end. Then it calls runcmd for the left end of thepipeline and runcmd for the right end, and waits for the left and the right ends to finish, by calling wait twice. The right end of the pipeline may be a command that itself includes a pipe (e.g., a | b | c), which itself forks two new child processes (one for band one for c). Thus, the shell may create a tree of processes. The leaves of this treeare commands and the interior nodes are processes that wait until the left and rightchildren complete. In principle, you could have the interior nodes run the left end ofa pipeline, but doing so correctly would complicate the implementation.

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

```
echo hello world | wc
```

could be implemented without pipes as

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

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

File system

The xv6 file system provides data files, which are uninterpreted byte arrays, anddirectories, which contain named references to data files and other directories. Xv6implements directories as a special kind of file. The directories form a tree, starting ata special directory called the root. A path like /a/b/c refers to the file or directorynamed 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 codefragments open the same file (assuming all the directories involved exist):

```
chdir("/a");

chdir("b");

open("c", O_RDONLY);



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

The first fragment changes the process's current directory to /a/b; the second neitherrefers to nor modifies the process's current directory.

There are multiple system calls to create a new file or directory: mkdir creates anew directory, open with the O_CREATE flag creates a new data file, and mknod createsa 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'smetadata 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 kerneldevice implementation instead of passing them to the file system.

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

```
#define T_DIR 1 // Directory

#define T_FILE 2 // File

#define T_DEV 3 // Device

struct stat {

  short type; // Type of file

  int dev; // File system's disk device

  uint ino; // Inode number

  short nlink; // Number of links to file

  uint size; // Size of file in bytes

};
```

A file's name is distinct from the file itself; the same underlying file, called an inode, can have multiple names, called links. The link system call creates another filesystem name referring to the same inode as an existing file. This fragment creates anew 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 inodeis identified by a unique inode number. After the code sequence above, it is possible todetermine that a and b refer to the same underlying contents by inspecting the resultof fstat: both will return the same inode number (ino), and the nlink count will beset to 2.

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

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

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

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

unlink("/tmp/xyz");
```

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

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

One exception is cd, which is built into the shell (8016). cd must change the current working directory of the shell itself. If cd were run as a regular command, thenthe shell would fork a child process, the child process would run cd, and cd wouldchange the child's working directory. The parent's (i.e., the shell's) working directory would not change.

Real world

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

Modern kernels provide many more system calls, and many more kinds of kernelservices, than xv6. For the most part, modern Unix-derived operating systems havenot followed the early Unix model of exposing devices as special files, like the consoledevice file discussed above. The authors of Unix went on to build Plan 9, which applied the ‘‘resources are files’’ concept to modern facilities, representing networks,graphics, and other resources as files or file trees.

The file system abstraction has been a powerful idea, most recently applied tonetwork resources in the form of the World Wide Web. Even so, there are other models for operating system interfaces. Multics, a predecessor of Unix, abstracted file storage in a way that made it look like memory, producing a very different flavor of interface. The complexity of the Multics design had a direct influence on the designers ofUnix, who tried to build something simpler.

This book examines how xv6 implements its Unix-like interface, but the ideas andconcepts 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 interprocess communication. After studying xv6, you should beable to look at other, more complex operating systems and see the concepts underlying xv6 in those systems as well.