Chapter 1

Operating system interfaces

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

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

This book uses a single operating system as a concrete example to illustrate operating system concepts. That operating system, xv6, provides the basic interfaces introduced by Ken Thompson and Dennis Ritchie's Unix operating system [15], as well as mimicking Unix's internal design. Unix provides a narrow interface whose mechanisms combine well, offering a surprising degree of generality. This interface has been so successful that modern operating systems—BSD, Linux, macOS, Solaris, and even, to a lesser extent, Microsoft Windows—have Unix-like interfaces. Understanding xv6 is a good start toward understanding any of these systems and many others.

As Figure 1.1 shows, xv6 takes the traditional form of a *kernel*, a special program that provides services to running programs. Each running program, called a *process*, has memory containing instructions, data, and a stack. The instructions implement the program's computation. The data are the variables on which the computation acts. The stack organizes the program's procedure calls. A given computer typically has many processes but only a single kernel.

When a process needs to invoke a kernel service, it invokes a *system call*, one of the calls in the operating system's interface. 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 hardware protection mechanisms provided by a CPU[1] to ensure that each

[1]This text generally refers to the hardware element that executes a computation with the term *CPU*, an acronym for central processing unit. Other documentation (e.g., the RISC-V specification) also uses the words processor, core, and hart instead of CPU.

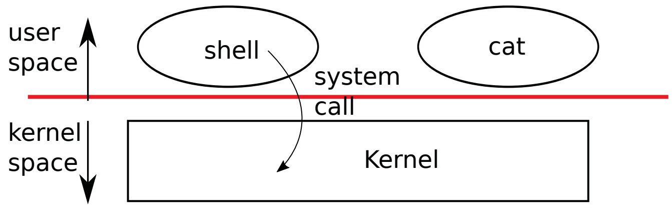

Figure 1.1: A kernel and two user processes.

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

The collection of system calls that a kernel provides is the interface that user programs see. The xv6 kernel provides a subset of the services and system calls that Unix kernels traditionally offer. Figure 1.2 lists all of xv6's system calls.

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

The shell is an ordinary program that reads commands from the user and executes them. The fact that the shell is a user program, and not part of the kernel, illustrates the power of the system call interface: there is nothing special about the shell. It also means that the shell is easy to replace; as a result, modern Unix systems have a variety of shells to choose from, each with its own user interface and scripting features. The xv6 shell is a simple implementation of the essence of the Unix Bourne shell. Its implementation can be found at (user/sh.c:1).

1.1 Processes and memory

An xv6 process consists of user-space memory (instructions, data, and stack) and per-process state private to the kernel. Xv6 *time-share*s processes: it transparently switches the available CPUs among the set of processes waiting to execute. When a process is not executing, xv6 saves its CPU registers, restoring them when it next runs the process. The kernel associates a process identifier, or `PID`, with each process.

A process may create a new process using the `fork` system call. `Fork` gives the new process exactly the same memory contents (both instructions and data) as the calling process. `Fork` returns in both the original and new processes. In the original process, `fork` returns the new process's PID. In the new process, `fork` returns zero. The original and new processes are often called the *parent* and *child*.

For example, consider the following program fragment written in the C programming language [6]:

```
int pid = fork();
```

System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

Figure 1.2: Xv6 system calls. If not otherwise stated, these calls return 0 for no error, and -1 if there's an error.

```
if(pid > 0){
  printf("parent: child=%d\n", pid);
  pid = wait((int *) 0);
  printf("child %d is done\n", pid);
} else if(pid == 0){
  printf("child: exiting\n");
  exit(0);
} 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. Exit takes an integer status argument, conventionally 0 to indicate success and 1 to indicate failure. The `wait` system call returns the PID of an exited (or killed) child of the current process and copies the exit status of the child to the address passed to wait; if none of the caller's children has exited, `wait` waits for one to do so. If the caller has no children, `wait` immediately returns -1. If the parent doesn't care about the exit status of a child, it can pass a 0 address to `wait`.

In the example, the output lines

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

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

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

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

The `exec` system call replaces the calling process's memory with a new memory image loaded from a file stored in the file system. The file must have a particular format, which specifies which part of the file holds instructions, which part is data, at which instruction to start, etc. Xv6 uses the ELF format, which Chapter 3 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 element of the argument array, 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` (user/sh.c:145). The main loop reads a line of input from the user with `getcmd`. Then it calls `fork`, which creates a copy of the shell process. The parent calls `wait`, while the child runs the command. For example, if the user had typed "`echo hello`" to the shell, `runcmd` would have been called with "`echo hello`" as the argument. `runcmd` (user/sh.c:58) runs the actual command. For "`echo hello`", it would call `exec` (user/sh.c:78). 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` (user/sh.c:145).

You might wonder why `fork` and `exec` are not combined in a single call; we will see later that the shell exploits the separation in its implementation of I/O redirection. To avoid the wastefulness of creating a duplicate process and then immediately replacing it (with `exec`), operating kernels optimize the implementation of `fork` for this use case by using virtual memory techniques such as copy-on-write (see Section 4.6).

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.

1.2 I/O and File descriptors

A *file descriptor* is a small integer representing a kernel-managed object that a process may read from or write to. A process may obtain a file descriptor by opening a file, directory, or device, or by creating a pipe, or by duplicating an existing descriptor. For simplicity we'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. We'll refer to input and output as *I/O*.

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 (user/sh.c:151), 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 file descriptor `fd`, copies them into `buf`, and returns the number of bytes read. Each file descriptor that refers to a file has an offset associated with it. `Read` reads data from the current file offset and then advances that offset by the number of bytes read: a subsequent `read` will return the bytes following the ones returned by the first `read`. When there are no more bytes to read, `read` returns zero to indicate the end of the file.

The call `write(fd, buf, n)` writes `n` bytes from `buf` to the file descriptor `fd` and returns the number of bytes written. Fewer than `n` bytes are written only when an error occurs. Like `read`, `write` writes data at the current file offset and then advances that offset by the number of bytes written: each `write` picks up where the previous one left off.

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

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

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