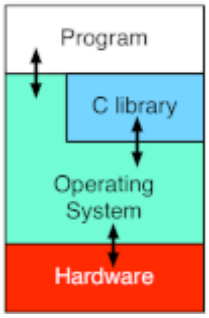
Introduction to the Operating System

Most interesting interactions with "the world"—reading input from the user, writing a file on disk, sending data over a network, etc.—require access to hardware devices (the keyboard, the disk drive, the network card), in ways that normal programs cannot perform themselves. One key aspect of this issue is that "normal" programs cannot be trusted to access hardware directly. If your program could read the disk directly, then it could ignore the permissions system in place to protect different users' files, and read or write any data it wanted. Furthermore, an error in a program that can access the disk directly could corrupt the entire file system, destroying all data on the system.

Instead, the program asks the *operating system* (OS)—low-level software responsible for managing all of the resources on the system for all programs—to access hardware on its behalf. The program makes this request via a *system call*—a special type of function call that transfers control from the program into the operating system. The OS checks that the program's request is within the bounds of its permissions before performing it, which is how the system enforces security rules, such as file permissions. If the request is not permissible, the OS can return an error to the program. If, on the other hand, the request is fine, then the OS performs the underlying hardware operations to make it happen, and then returns the result to the program.



The figure above depicts the conceptual relationship between the program, C library, operating system, and hardware. While your C code *can* make system calls directly, it is more common to use functions in the C library. The C library functions then make system calls as they need. The OS then interacts with the hardware appropriately.

For example, suppose you call **printf** to write a string to standard output. The **printf** function is part of the C library, and involves significant code that does not require system calls: it must scan the format string for % signs, and perform the appropriate format conversions, building up the actual string to print. At some point, however, **printf**must actually write the resulting string out to standard output, which requires a system call. To perform this task, **printf**uses the **write** system call.

Novice programmers are often imprecise about the difference between a system call (which transfers control into the OS, requesting it to perform a task), and a library call (which calls a function found in a library, such as the C standard library). For example, calling **printf**a "system call" is technically incorrect, though most people will understand what you mean. Being precise with this distinction is useful for two reasons. First, the more precise you are with your terminology, the more knowledgeable you come across during interviews (if you are interviewing for a programming-related job). Second, if you

need to look a function up in the man pages, knowing whether it is a system call, or part of the C library tells you which section to look in—system calls are found in section 2, while C library functions are found in section 3.

Completed