Other Interactions

coursera.org/learn/interacting-system-managing-memory/supplement/PaOxQ/other-interactions

Sometimes programs interact with the rest of the system or outside world in ways other than reading from or writing to files. UNIX tries to make as many of these access types as are reasonable appear to be similar to accessing files—presenting the same interface, and thus allowing the use of familiar functions to perform these operations. For operations conforming to this model, the underlying system call to initiate access returns a file descriptor, which is then passed to other appropriate system calls (such as **read** or **write***)*. If the user wants to use the IO functions from **stdio** (**fprintf**, **fgets**, **fgetc**,...), she can use the **fdopen** library call to get a **FILE** * corresponding to the file descriptor.

One example of an interaction that "looks like a file" is reading input from the user and printing it to the terminal. You have already printed things to the terminal with **printf**, which prints the output to **stdout**, a **FILE** * that is connected to "standard output." By default, standard output is the terminal; however, as you learned in Course 2, you can redirect the output so that it goes to an actual file instead. You can also use **fprintf** to print to **stderr**, which is another **FILE** * that also goes to the terminal by default. While **stdout** and **stderr** both print to the terminal by default, they serve different purposes: one is for output and the other is for errors.**[1]** You can also read from the terminal to get input from a user by reading from **stdin**. Each of these **FILE** *s is declared in **stdio.h**. They are all open when your program starts (the C library opens them before **main**), and they correspond to file descriptors 0 (**stdin**), 1 (**stdout**), and 2 (**stderr**). You should not ever close these **FILE** *s, as the C library is responsible for them, not your code.

Another example of an interaction that "looks like a file" is transferring data across the network. The program obtains a file descriptor for a *socket* — the logical abstraction over which data is transferred—via the**socket** system call. Depending on the network protocol the program desires, additional setup may then be required to specify the destination, establish the connection, etc. However, once the socket is set up, data can be sent across the socket by using the write system call, or reading from the network by using the read system call (which, if no data has been received, will *block*—cause the program to wait—until data arrives). Of course, if the fdopensystem call has been used to setup a FILE *, then library functions like **fprintf**or **fgets**can be used (which make write and read system calls on the underlying file descriptor).

Another form of interaction that looks like a file is a *pipe*. The name "pipe" may sound familiar from the similarly named shell construct (as in **cmd1 | cmd2** at the shell), which uses this type of communication. A pipe is a one-way communication channel between two processes. One process writes data into one "end" of the pipe, and the other process reads from the pipe, obtaining the data that was written in by the first process (or blocking if no data is available). This communication looks exactly like reading/writing a file to each process (as they again,read/writethe file descriptors, or use library functions that do so). The shell uses this when you use the pipe operator, as it sets up a pipe, and

arranges for the standard output file descriptor of the first process to be the writing end of the pipe, and the standard input file descriptor for the second process to be the reading end of the pipe.

Another way that UNIX provides access to things that are not traditional files in a way that looks like files is through *device special files*. These appear as files in the file system (you can see them with ls, for example), but have a special file type indicating that the OS should not attempt to read/write data from/to the disk (or other media), but should instead perform special functionality. These files are typically found in the */dev* directory.

For example, on Linux, the "file" */dev/random* provides access to the secure pseudo-random number generator provided by the kernel. You can open this file for reading with **fopen** (or **open**), and read from it, with whatever method you prefer. However, when you perform a read on this file, the kernel recognizes that it should perform a random number generation routine to supply the data, rather than reading the disk. The kernel will generate random numbers according to its algorithms and return that as the data read by the system call. The read operation may block if the kernel's entropy model indicates it needs more entropy to generate numbers securely.**[2]** The */dev/urandom* device can be used instead to generate numbers with the same algorithm, but without regard to if there is sufficient entropy available for security-sensitive purposes.

There are, however, things that do not fit into the "everything is a file" model. These are typically handled by system calls, but may not involve file descriptors. For example, if you need your program to determine the current time, you can use the **gettimeofday** system call, which just returns the time of day. There are also system calls to create a new process (**fork***)*, replace the currently running program with another (**execve**), exit the current program (**exit**is the library call typically used for this purpose, **_exit**provides access to the underlying system call, which is rarely needed), and many more things. In fact, there are a few hundred system calls. As you gain experience programming, they will become more familiar to you.

One other form of interaction is when the OS needs to inform the program of something *asynchronously*—not at a time that the program explicitly asks for it, but rather at any time during its execution. Here, unlike with a system call, the OS is initiating the communication with the program. UNIX-style OSes support many signals (each has a number, and a symbolic name, indicating what it represents).

Most signals are fatal to the program by default—if the OS needs to deliver a fatal signal to the program, it will kill the program in question. We have already seen an example of one fatal signal—although we have not discussed it as a signal. When your program segfaults (which happens when your program accesses memory in certain invalid ways), the OS sends it SIGSEGV, which kills the program. Some other signals have a default action of being ignored, in which case nothing happens if the OS delivers that signal.

Programs can change what happens for each particular signal (except for signal 9, which is SIGKILL, which is always fatal). One option for the new behavior of the signal is to have the OS cause the program to run a particular function (the program specifies which

function when it makes the system call asking the OS to modify its behavior for that signal). We are not going to go into the details of signals, signal handling, and related topics here, but you should at least know they exist, and anticipate learning about them in the future. We will note that setting your program to just ignore SIGSEGV because you cannot get it to work (and it keeps segfaulting) is a Bad Idea—remember you never want to hide an error, you always want to fix it.

[1] This distinction can be quite useful if you want to redirect one but not the other: for example, you might want to write the output to a file but see any errors immediately on the terminal while you run the program.

[2] The kernel typically generates entropy by tracking hard-to-predict information from events it can observe (such as the number of microseconds between hardware events the kernel must deal with, like key presses, mouse movements, or data arriving over the network interface), which is then the basis for pseudorandom number generation. As numbers are generated, it may become possible to guess information about the pattern without new hard-to-predict events to feed the algorithm, so the kernel waits for more such events before supplying more numbers.

Completed