Learning Objectives

- Describe the use of Unix I/O library functions.
- Understand when short reads and writes happen and how to handle them.
- Recognize the implementation differences and resulting performance differences between Unix I/O and Standard I/O.
- Predict the behavior of multi-process programs when file descriptors are opened and copied (with `dup` and `dup2`).
- Describe how shell I/O redirection is implemented in terms of Unix I/O operations.

Getting Started

The directions for today's activity are on this sheet, but they refer to programs that you'll need to download. To get set up, run these commands on a shark machine:

```
$ wget http://www.cs.cmu.edu/~213/activities/system-io.tar
$ tar xf system-io.tar
$ cd system-io
$ make
```

1 Unix I/O

The Unix I/O API consists of low-level functions for opening, closing, reading from, and writing to files. Open files are represented in this API by *file descriptors*, which are non-negative integers.

The file descriptors 0, 1, and 2 are already open when a program starts up. They correspond to the Standard I/O streams standard input (stdin), standard output (stdout), and standard error (stderr) respectively.

Examine the program `unixcat.c`, which opens a file, reads its contents, and writes its contents to stdout.

Problem 1. When you're ready, try using this program to print its own source:

```
$ ./unixcat unixcat.c
```

What went wrong? Edit `unixcat.c` to fix the problem (hint: it's on the line after the `FIXME` comment); run `make` again to recompile, then try using it again. What did you have to do to make it work correctly?

The `unixcat` command does not print the complete file due to a **short count**; bytes were read from the file, but fewer than `BUFFER_SIZE`. Changing the condition of the `while` loop to > `0` makes the program work correctly.

2 Standard I/O

The C standard does not specify the Unix I/O functions. It does specify the *standard I/O* functions, which operate on files at a higher level. (These are implemented using the Unix I/O functions on Unix systems.) Some of the standard I/O functions are very similar to Unix I/O functions, such as `fread` and `read`. Others, such as `printf`, provide facilities for complex "formatted" I/O. All of the standard I/O functions represent open files with `FILE` objects, which are `structs` containing file descriptors but also various other data.

One of the most important reasons to use standard I/O is *buffering*: data can be temporarily stored inside each `FILE` (in a `char` array) to reduce the number of calls to `read` and `write` that are necessary. To see how buffering affects a program's output, look at the program `three.c`. It prints three three-word phrases using different combinations of `printf` and `write`. When you're ready, run the program with `./three`.

Problem 2. Which of the three phrases were printed in the same order as they appear in the source code? Why did this occur? What is different about the way the three phrases were printed that caused this?

In this program, `stdout` is configured to buffer data until a complete line can be printed. The first phrase is printed out of order because the first call to `printf` does not print a complete line, so the word "believe" is held in the buffer. `write` bypasses the buffer and prints "in". The second call to `printf` completes the line, and "believe yourself!" is printed all at once, after "in".

The second phrase is printed in order because each string passed to `printf` is a complete line (ending with a `'\n'` character) so there is nothing held in `stdout`'s buffer between calls.

The third phrase is printed in order even though its first call to `printf` does not supply a complete line, because `fflush(stdout)` causes the contents of the buffer to be printed immediately.

Problem 3. You can use the `strace` program to examine the system calls made by `three`. Run it like this:

```
$ strace -e trace=write ./three > /dev/null
```

What do you notice about the calls to `write`? Does this agree with what you observed?

Each call to `write` is visible. Each call to `printf` corresponds to an additional call to `write`, except for the first phrase, where the write for "in" happens before the buffered write for "believe yourself". This is consistent with what we observed earlier.

Note: In this problem, `strace` was directed to report *only* calls to `write`, and I/O redirection was used to suppress the output directly from `three`. You might be curious to see what happens if you simply run

```
$ strace ./three
```

(This will produce a lot of output—be prepared to scroll back.)

2.1 Buffering and Performance

Buffered I/O aims to increase efficiency by reducing the number of calls to `read` and `write`, which have a lot of overhead (tens of thousands of clock cycles). Examine the program `timing.c`, which measures the time it takes to write data to `/dev/null` one byte at a time, using both Unix I/O (`write`) and Standard I/O (`fputc`). (`/dev/null` is a special file called the "null device," or, colloquially, the "bit bucket," which discards anything written to it. We've already used it once in this activity.)

Problem 4. The program `timing` takes one command-line argument, the number of bytes to write. When you are done looking over its code, try having it write just one byte each way:

```
$ ./timing 1
```

Which way is faster, `write` or `fputc`? Is that what you expected? You may need to run it several times to see a pattern.

If we are only writing a single byte, it is faster to use `write`. This is because of the overhead of setting up the `FILE` and copying data into the buffer before calling `write`. (Notice that we are measuring the cost of opening and closing the file using each API, not just the writes.)

Problem 5. Try increasing the number of bytes written, in steps. How many bytes do you need to write before buffering is faster? If you make the number of bytes larger and larger, what do you think the *asymptotic* performance curves look like?

Buffering starts to be worth the setup overhead at something like 100 bytes written. Asymptotically, both are linear in the number of bytes written, but the slope of the line is much steeper for `write`.

Problem 6. Based on what you have just observed, when do you think you should use Unix I/O functions, and when do you think it will be better to use Standard I/O?

The lower-level Unix functions are normally chosen when it is necessary to have precise control over what system calls happen when, or when Standard I/O is unsafe (e.g. in a signal handler). They can also be a good choice for reading or writing large blocks of data *all at once*. Standard I/O works better when you are doing many small reads and writes (e.g. to process a file character by character or line by line).

Problem 7. (advanced) Edit `timing.c`. On the line that reads

```
setvbuf(stream, NULL, _IOFBF, 0); // Buffer in large chunks.
```

change `_IOFBF` to `_IOLBF`. Recompile (`make`). Run `timing` again, with a large argument (say, 100000). What changed? Why might that have happened?

Changing `_IOFBF` to `_IOLBF` tells the Standard I/O functions to buffer only one line of output instead of large chunks of data. The value of the variable `c` is `'\n'`, so every call to `fputc` completes a line of output. Effectively, the Standard I/O functions are no longer doing any buffering and are now *slower* than `write`.

3 File Descriptors, Fork, and Dup2

Each time the `open` function is called, a new *open file table entry* is created and the file descriptor corresponding to that entry is returned. However, the `dup` and `dup2` functions can be used to *duplicate* a file descriptor. Duplicated file descriptors point to the *same* open file table entry. `dup` takes the old file descriptor as an argument and returns the duplicate. `dup2` takes both the old and new descriptors as arguments, and makes the new descriptor be a duplicate of the old. If the new descriptor was open already, it is closed first (atomically).

Examine the program `doublecat.c`, paying particular attention to the `print2` function, which takes in two file descriptors and prints each file, one character at a time.

Problem 8. The file `abcde.txt` contains the characters `abcde`, followed by a newline character. If you were to run `$ ./doublecat abcde.txt` what would be printed in each case?

Case 1: abcde. Case 2: aabbccddee. Case 3: abcde.

Problem 9. Run the program as suggested (`$ ./doublecat abcde.txt`). Were your predictions correct? Did the output differ in the three cases? Why do you get the output you do?

In cases 1 and 3, there is only one open file table entry and therefore only one *file position*, so the characters printed are `a b c d e \n`, and they alternate between `fd1` and `fd2`. In case 2 there are two independent open file table entries so each character is read twice, once from each file descriptor.

To further complicate matters, child processes share the open file descriptors of their parents. (It is as if `fork` calls `dup` for each descriptor—but the new descriptors go into the child's file descriptor table and have the same numbers that they did in the parent.)
Examine the program `childcat.c`, which forks two child processes, each of which print two letters from a shared file descriptor, while the parent prints one letter from that file.

Problem 10. If you run `childcat` on `abcde.txt`, what could be printed? Take a moment and write your guess below. Then, run `$ ./childcat abcde.txt`. Did the output match what you expected?

The output is very likely to be `abcde`. However, due to scheduling nondeterminism, it's possible for these letters to appear in a different order.

Problem 11. Run `$ ./childcat abcde.txt` several more times. Does it *always* print the same thing? Look at the code carefully. Is it *guaranteed* to print the same thing always? If it isn't, are there any constraints on what it can and cannot print?

You might see it always printing the same thing, but it isn't guaranteed to. The child and the grandchild each read (and then print) two characters from the file. Those pairs of characters will appear in the same order that they appeared in the file, but there might be other characters in between them, and neither the child nor the grandchild is guaranteed to go first. The character read and printed by the parent can appear at any point in the output. The newline printed by the parent will always be the last thing `childcat` prints. These are the only constraints.

4 Shell I/O Redirection

Shell I/O redirection is a tool for defining the input and output of shell commands in a high-level, uniform manner.

Problem 12. Try running the command

```
$ /bin/echo 15213 rocks > phrase.txt
```

This writes the string "15213 rocks" to the file `phrase.txt`, creating it if it does not exist. What Unix I/O and process control functions do you expect the shell will use to run this command?

The shell will call `fork` to create a child process. The child process will then call `open` on `phrase.txt`, for writing (not appending), followed by `dup2` to assign the newly opened file as standard output, and finally `execve` to run the program `/bin/echo`. In the meantime, the parent shell process calls `waitpid` to reap the child process once it completes.

Problem 13. We can use `strace` to observe how the shell runs this command. Run it like this:

```
$ strace -f -e trace=process,open,dup2,write \
    /bin/sh -c '/bin/echo 15213 rocks > phrase.txt'
```

Does the series of operations you observe, match what you expect?

Yes, for the most part. There are some small differences, such as `clone` instead of `fork` and `wait4` instead of `waitpid`. This is normal—`strace` sees details that the C library normally hides. (If you looked at the code for `/bin/sh` you would probably see a call to `fork`.) Note the way `open` is called, with flags that open the file for writing only (`O_WRONLY`), creating it if it does not exist (`O_CREAT`), and erasing its contents if it does (`O_TRUNC`). (Run the command `man 2 open` for more information.)

Problem 14. Examine the `phrase.txt` file. Did the traced invocation change its contents? Now try the following command (untraced):

```
$ /bin/echo 15213 rocks >> phrase.txt
```

What did that do to the file? What do you think the shell did differently because you used >>?

The traced invocation overwrote the existing file with the same contents, so it didn't appear to change. This new invocation *appends* to the file, instead of overwriting it. The shell achieves this by passing different flags to `open`: `O_APPEND` instead of `O_TRUNC`.

In addition to redirecting standard input and output to/from files, the shell has the ability to *pipe* the output of one program to the input of another. You request this with the `|` operator.

Problem 15. An example of a command that uses pipes is

```
$ ps aux | grep $USER
```

This runs `ps aux`, which prints out a report on all the running processes on the current machine, and sends the output to the input of `grep $USER`, which searches for lines containing your username and prints only those lines. The output of `grep` goes to your terminal. (`$USER` is a *shell variable*. The shell will replace it with your actual username when it runs the `grep` program.)

To run this "pipeline", what Unix I/O and process control functions will the shell call, and in what order? How does this differ from the calls done for simple redirection to file? (One of the functions that's needed, we haven't mentioned at all yet. `man 2 pipe`.)

The shell will need to `fork` two processes, one for `ps` and one for `grep`, and it will need to set up communication between the two processes so that `write` operations done by `ps` make data available for `grep` to read. "Pipe" is not just the name of this shell feature, it's the name of the communication channel that is used, and the name of the system call that creates it. Before forking, the shell will call `pipe` once, which will open *two* file descriptors, one for reading and one for writing. It will `dup2` the write fd to `ps`'s standard output and the read fd to `grep`'s standard input. Then it will close all unnecessary fds in each child and `execve` the appropriate program in each. Only after *all* of the child processes have been created will it call `waitpid` for any of them. (Why do you suppose it has to do it that way?)