15213 Lecture 16: System-level I/O

Learning Objectives

- Describe the use of Unix I/O library functions and short counts.
- 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`).
- Enumerate the order, number, and nature of Unix I/O calls made by a shell in I/O redirection.

Getting Started

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

1. `$ wget http://www.cs.cmu.edu/~213/activities/lec16.tar`
2. `$ tar xf lec16.tar`
3. `$ cd lec16`

1 Unix I/O

The Unix I/O library provides several low-level functions for opening, closing, reading from, and writing to files, which are represented as integer **file descriptors**. Recall that 0, 1, and 2 are the file descriptors for 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. When you're ready, build all the programs by typing: `$ make`.

1. Try using the resulting `unixcat` executable to print its source: `$ ./unixcat unixcat.c`. What went wrong? Try fixing the line after the `FIXME` comment and running `$ make` again. Does this fix the problem?
   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` will fix the problem.

2 Standard I/O

The C standard library I/O functions, on the other hand, provide some higher-level functions to interact with files. It includes wrappers for Unix I/O functions, such as `fread` and `fwrite` for the Unix I/O `read` and `write` functions. The C standard library I/O functions also include some additional functions for more sophisticated reading and writing, such as `fgets`, `fputs`, `fscanf`, and `fprintf`. However, open files are represented as **streams**, abstractions for a read/write buffer in addition to a file descriptor.

To see the difference between the buffered Standard I/O functions and the unbuffered Unix I/O functions, look at the program `three.c` that prints three three-word phrases. When you're ready, run the program with `$ ./three`.

2. Did the three phrases print correctly in the order you expected? Why did this behavior occur? What is different about the way the three phrases were printed that caused this?
   The first phrase was printed incorrectly because the Standard I/O `printf` function is buffered and delays writes while the Unix I/O `write` is not and immediately writes to `stdout`. Ending each `printf` with a newline character, or calling `fflush(stdout)` flushes the `printf` buffer to `stdout`.
3. You can use the Linux `strace` program to examine the system calls made by `three` by running `$ strace ./three`. Look at the last few lines of output. What do you notice about the calls to `write`? Does this agree with what you observed?
   There is a separate call to `write` for each `printf` and `write` except for the first phrase, where the write for "in" happens before the buffered write for "believe yourself".

Buffered streams aim to increase efficiency of reads and writes by reducing the number of calls to the underlying Unix I/O `read` and `write` functions, each of which require expensive (> 10,000 clock cycles) Unix kernel calls. We will now explore the performance differences between the buffered, higher-level Standard I/O functions and the low-level Unix I/O functions. Examine the program `timing.c`, which times writes of a specified number of lines to `/dev/null`[1] using both `write` and `fprintf`.

4. After running the program (`$ ./timing`), it first prompts you for the byte to be written, then the number of bytes to write to `/dev/null`. Try writing a single byte by entering `a`, then 1. Is `write` or `fprintf` faster? Is this what you expected?
   The Unix I/O `write` is faster than `fprintf`, because of the overhead that `printf` incurs due to setting up its buffer, writing to its buffer, and other overhead from setting-up higher-level features.

5. Now try writing a large number of lines (a good number to try is 1,000,000). Does `write` or `fprintf` run faster? Why?
   This time, the Standard I/O `fprintf` runs faster because the buffering of the Standard I/O library results in fewer (expensive) kernal calls to `write`, which gives asymptotically higher performance.

6. Based on what you have just observed, what situations would it be appropriate to use Unix I/O functions like `write`? What situations would be better suited to Standard I/O functions like `fprintf` (or `printf`)?
   It may be appropriate to use the lower-level Unix I/O functions in cases where you need the absolute highest performance per write call or where async-signal-safety is needed. The Standard I/O functions are appropriate when you don't want to write tricky error-handling (like accounting for short counts) or will be doing many reads or writes and want better asymptotic performance.

7. (advanced) Thinking back to what you observed in the previous section, `printf` (and `fprintf`) seem to flush the print buffer when reading a newline `n` character. Try writing a million or so newline characters by pressing return without entering a character. Is `fprintf` faster than `write` in this case? Why might this be?
   The C standard states that `printf` and `fprintf` need only flush on a newline if the output device is an interactive one (like `stdout`), and `/dev/null` is not defined as an interactive device. Therefore, `printf` and `fprintf` are fully-buffered when writing to files. Interestingly, what constitutes an interactive device is implementation-defined. You can see the manpage for `setbuf` for more information on different buffering styles (line-buffering versus character-buffering).

[1] `/dev/null` is a special Linux file called the **null device** (or, colloquially, the bit-bucket or the blackhole) that discards anything written to it while reporting that the write succeeded.

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. `dup` takes the old file descriptor and returns the new file descriptor while `dup2` takes the old file descriptor and the new file descriptor and makes the new file descriptor a copy of the old file descriptor, closing the new file descriptor if necessary. Duplicated file descriptors point to the same open file table entry.

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.

8. The file `abcde.txt` contains the string `"abcde"`. Before running the program, take a moment to think about what the program will print in each of the three cases. What do you think will be printed in each case?
   Case 1: abcde. Case 2: aabbccddee. Case 3: abcde.

9. Now run the program with `$ ./doublecat abcde.txt`. Were your predictions correct? Did the output differ in the three cases? Why?
   Yes. The output is different when calling `open` twice in Case 2, because there are two independent open file table entries, while in Cases 1 and 3, there is only one open file table entry being read from.

To further complicate matters, child processes share the open file descriptors of their parents. 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).

10. What do you think could be printed from the file `abcde.txt`? Take a moment and write your guess below. Then, run `$ ./childcat abcde.txt`. Did the output match what you expected?
    `abcde` could be printed. In fact, any series of five letters can be printed due to scheduling nondeterminism.

11. While nondeterministic execution due to process scheduling is unlikely to occur in a program this small, it remains possible. Look at the code carefully. Briefly describe the strings that CANNOT be printed by `childcat`, if any. If there are none, explain why.
    There are no strings that cannot be printed, because while child 0 and child 1 both print two characters, they read them one at a time instead of two at a time.

4 Shell I/O Redirection

Shell I/O redirection is a useful tool for reading input to a command from a file or writing the output of commands to a file.

12. Try running the command `$ /bin/echo 15213 rocks > phrase.txt`, which 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 call to run this command?
    The shell will likely call `fork` to create a child process, then `open` on the output file for writing (not appending), then `dup2` to redirect the file descriptor of standard output to the file descriptor corresponding to the opened file, then `execve` to run the `/bin/echo` program.

The function tracing library from last lecture has been extended to also trace the relevant function calls. Start a traced shell by running `$ LD_PRELOAD=./libtrace.so sh`.

13. Try rerunning the command inside the traced shell. Did the series of Unix I/O library functions match what you expected?
    Yes. Note the flags passed to `open` (`O_WRONLY`, `O_CREAT`, and `O_TRUNC`), which open the file for writing only, creating the file if it does not exist, and truncating the size of the file to 0 (`$ man 2 open` for more info).
14. Now exit the traced shell and examine the `phrase.txt` file. Did the second invocation change its contents? Now try the following command (untraced): `$ /bin/echo 15213 rocks >> phrase.txt`. Reexamining the file and the first trace, what do you expect the shell does differently to achieve this new behavior?
    The second invocation did not change its contents, because it overwrote the existing file. The shell passes different flags to `open` to append to the file instead of overwriting it (specifically, `O_APPEND` instead of `O_TRUNC`).
15. (advanced) You may have noticed that the traced shell calls `open` a few times when starting up. However, it does not always call `open` when exited. What is the purpose of each `open` call on the shell start-up? What do you think `/dev/tty` is? What is in the file `/etc/inputrc/`? And what are the conditions under which `open` is called upon exiting the shell?
    The `open` calls on shell start-up correspond to (1) opening the (character) device "file" corresponding to the current process's terminal, (2/3) opening a user's `.bash_history`, which contains the history of commands previously ran by the user for up-arrow command completion, and (3) and opening a shell configuration file (`/etc/inputrc`) for key-mappings and other settings. `open` is called upon exiting the shell when one or more commands have been run (and therefore `.bash_history` must be updated).

While the `<` and `>` characters are for redirecting to/from a file or a stream, the `|` character can be used for a more general I/O redirection, such as "piping" the output of one command to be the input of another command.

16. (advanced) Consider the command `$ ps aux | grep 'MYANDREWID'`. `ps aux` prints out all the running processes on the current machine while `grep` searches for and prints out the lines that contain the specified string. 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?
    The shell will call `fork` twice (once for `ps` and once for `grep`), and one `dup2` for each, redirecting placeholder file descriptors 3 and 4 (opened at the start of the shell) to redirect `stdout` in `ps` to `stdin` in `grep`. Then, `ps` needs to read the metadata for each process, hence a great many `open` calls. This differs from the calls for redirection to a file in that the processes are being redirected to each other, instead of to an opened file.