Performance Analysis: System Calls vs. Standard I/O

Sean Balbale

November 26, 2025

Analysis of Empirical Results

A. For the smallest buffer size (e.g., 1 byte), which version is faster, and why?

In my testing, `copy2` (Standard I/O) is exponentially faster than `copy1` (System Calls) at the smallest buffer sizes. At 64 bytes, `copy1` took over 524,000 milliseconds (nearly 9 minutes), whereas `copy2` finished in roughly 7.6 seconds. At 1 byte, `copy1` failed to complete within a reasonable timeframe, while `copy2` finished in 9.3 seconds.

The reason for this discrepancy is the cost of context switching. `copy1` forces a system call (`read` and `write`) for every single byte processed. This requires the CPU to switch from user space to kernel space and back millions of times, creating massive overhead. `copy2`, however, utilizes the internal buffering provided by the `stdio` library. Even though I requested 1 byte, `fread` likely fetched a larger chunk (e.g., 4KB) into its internal buffer, allowing subsequent calls to be satisfied from memory without triggering a context switch for every byte.

B. How does performance change as the buffer size increases for both programs? At what buffer size do the performance gains start to diminish?

As the buffer size increases, the performance of `copy1` improves dramatically because the ratio of data payload to system call overhead becomes much more favorable. `copy2` shows much less variance because its internal buffering masks the inefficiency of small user buffers.

However, the gains begin to diminish significantly once the buffer size hits the 4KB to 64KB range.

- **4KB:** `copy1` (8497ms) vs. `copy2` (8377ms) — The gap closes.
- **64KB:** `copy1` (1642ms) vs. `copy2` (1669ms) — Both see a major jump in speed compared to 4KB.
- **1MB:** `copy1` (1152ms) vs. `copy2` (1192ms) — The improvement from 64KB to 1MB is present, but much smaller than the jump from 4KB to 64KB.

It seems that once the buffer is large enough to minimize the frequency of system calls (around 64KB), increasing it further yields diminishing returns as the bottleneck shifts from CPU overhead to actual disk I/O speed.

C. Does standard I/O always outperform system calls? Why or why not?

No, Standard I/O does not always outperform system calls. My results show that at the largest buffer size tested (1MB), `copy1` (1152ms) was actually slightly faster than `copy2` (1192ms).

This happens because Standard I/O introduces an extra layer of data copying. `fread` copies data from the kernel to its internal buffer, and then copies it again to the user's buffer. When using raw system calls with a sufficiently large buffer (like 1MB), `copy1` reads data directly from the kernel into the user buffer, skipping that intermediate copy. When the buffer is large enough that system call overhead is negligible, that extra memory copy in `copy2` becomes a slight liability.

D. What role does the user-space buffering provided by `fread`/`fwrite` play in the performance compared to repeated `read`/`write` system calls?

The user-space buffering in `fread`/`fwrite` acts as an optimization layer that aggregates many small I/O requests into fewer, larger system calls. It essentially decouples the logical read size (what my code asks for) from the physical read size (what the OS handles).

In the case of the 64-byte test, `copy1` had to interact with the kernel thousands of times more than `copy2`. The user-space buffer allows `copy2` to stay in user mode for most operations, only trapping into the kernel when the internal buffer runs dry. This explains why `copy2` remained relatively performant even when I set the explicit buffer size to 1 byte.

E. Suppose you were writing a high-performance file copying tool. In what situations might you choose pure system calls versus standard I/O?

If I were building a high-performance tool where I had complete control over the buffer size, I would choose pure system calls (`read`/`write`). As seen in the 1MB test, bypassing the standard library's internal buffering offers a slight performance edge by eliminating redundant memory copies, provided I can allocate a large enough buffer (e.g., 64KB or larger).

However, I would choose Standard I/O (`fread`/`fwrite`) if the application needed to handle unpredictable or very small read/write sizes, or if portability across different operating systems was a priority. Standard I/O is safer because it guarantees decent performance regardless of how the user code chunks the data, whereas system calls require the programmer to manually optimize the buffer size to avoid the massive performance penalties seen in my 64-byte test.