Editor: Diomidis Spinellis
dds@aueb.gr

Optimizing File Streaming Input/Output

Diomidis Spinellis, Athens University of Economics and Business and Delft University of Technology

PERFORMANCE IS A critical attribute of system software since even small improvements are amplified across the countless CPU instructions devoted to it. In the two previous installments of this column, I described how I ported the Unix *sed* stream editor[1] from C into Rust[2] and the system's design.[3] Here, I describe how I optimized its input/output (I/O) performance by exposing advanced operating system (OS) facilities as Rust abstractions.

Copy Galore

Rust's expressiveness and the existence of my earlier mature C implementation allowed me to risk a more ambitious implementation of I/O operations. Specifically, I employed memory mapped files and an abstraction for representing I/O data in them to allow the processing of files with little or no copying of data.

In a typical loop that copies lines from one file to another, such as the following, bytes are expensively copied at least four times between several buffers (see Figure 1).

```
for line in reader.lines() {
    let line = line?;
    writeln!(output, "{}", line)?;
}
```

Digital Object Identifier 10.1109/MS.2025.3621023
Date of current version: 23 December 2025

The description that follows applies to the Linux kernel, but other OSs operate in a very similar way.[4] First, the kernel's *readpage* function arranges for the input storage device data to be transferred to its page cache. In modern hardware this is often handled through the device's hardware without burdening the CPU using direct memory access (DMA): the storage device takes control of the memory bus to transfer the data directly to a specified memory area.

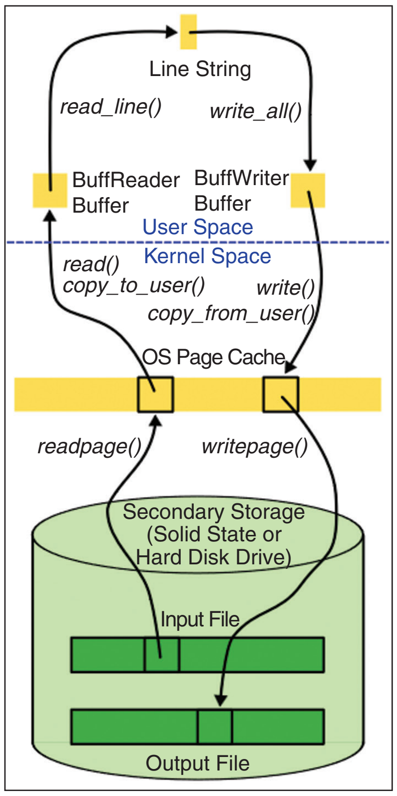


FIGURE 1. Typical file line processing.

Then, the *read* system call copies data from the OSs page cache (which resides in the OSs memory space) into the program's input buffer (which resides in the program's memory space). Because operating system calls are horrendously expensive, the program's input and output buffers are used to minimize the number of read and write calls.

The Rust's line reader then carves out a single line from the input buffer and copies it into a string. When the program outputs the line string, its data are written into an output buffer, from there they are copied to the kernel's page cache with the *write* system call, and from there the *writepage* function finally transfers them to the destination storage device.

When little other processing is done, the cost of copying can dominate the processing time. Can we avoid it?

Memory-Mapped I/O

Memory mapping is an I/O technique where a program instructs the OS to devote (map) a memory area where the program can access directly the file's contents. The OS uses its paging mechanism and the hardware's virtual memory support to bring the file's data to the mapped area on demand, bypassing read/write operations through the buffer cache and eliminating the cost of copying from it into an input buffer (see Figure 2).

Under this scheme, the mapped memory area is initially virtually mapped to correspond to the file as its backing storage, without actually reading the file's contents. When a program tries to access data in an unpopulated file-backed page, the hardware generates a page fault. The OS responds to the fault by reading the page's data from the file and then handles control back to the program. If the file is larger than the available memory, the OSs virtual memory mechanisms ensure that only the most useful file pages reside in main memory. Similar methods are also used for modifying memory-mapped files, but this is not the method I used for *sed*'s output because the size of the output file is unknown in advance and therefore it cannot be allocated and mapped.

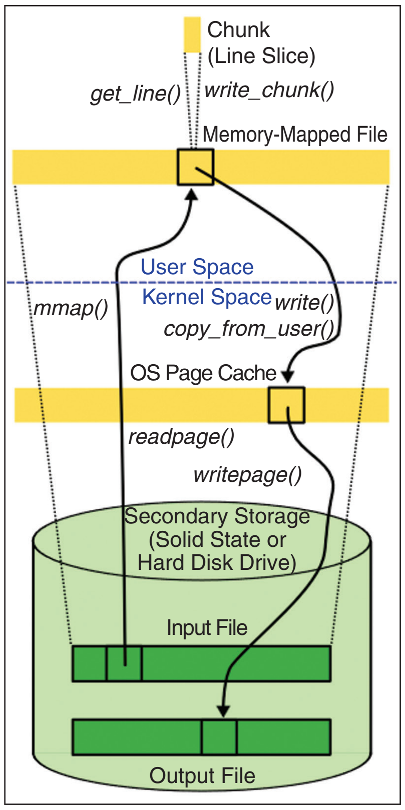


FIGURE 2. Memory-mapped I/O.

On top of the memory-mapped input file, I abolished the copy from the input buffer into a string, by creating a cursor (*MmapLineCursor*) that reads the file line-by-line, returning data as Rust slices of the memory-mapped region. (You can find the code on the command's repository: https://github.com/uutils/sed.)

Many input sources, such as pipes, devices, and network sockets, don't support memory mapping. To deal with this, I defined a Rust enum that provides interfaces for reading lines both from memory-mapped files and by using Rust's universally supported *read* function.

```
pub enum LineReader<'a> {
    MmapInput {
        mapped_file: Mmap,
        cursor: MmapLineCursor<'a>,
    },
    ReadInput(ReadLineCursor),
}
```

For speeding up the output, I avoided copying into Rust's *BuffWriter* I/O buffer by having the *write_chunk* function I created for output issue a direct OS *write* system call with a pointer to the memory-mapped data. Thus, when *sed* processes large parts of a file without modifications, the file's data never leave the OS kernel's memory space, saving the overhead of copying from and to it. To avoid the cost of issuing an expensive *write* system call for every output line, the *write_chunk* function, coalesces adjacent output data slices, saving them for later output with a single *write* call. Again here, if data are modified or come from nonmemory-mapped input, the routine flushes any pending output from the memory-mapped area and falls back to Rust's *BuffWriter* methods. With these abstractions in place, a line copying loop looks as the listing below; the chunk variable only contains a few bytes describing the data to copy, rather than the actual data.

```
while let Some(chunk) = reader.get_line()? {
    output.write_chunk(&chunk)?;
}
```

Compared to the typical *lines()* and *writeln()*-based I/O, the memory mapped one can be up to 2.8 times faster, with many I/O heavy processing tasks being 10 to 60% faster.

File Extent Magic

When writing this column, I realized that even when the input file wasn't modified, the OS kernel would still copy the input file's blocks to page cache blocks corresponding to the output file. Could this needless copy be eliminated?

It turns out that this last remaining copy can indeed be eliminated under Linux and file systems that support shared file extents. In file systems, such as XFS and BTRFS, which support this feature data can be shared among many files. Thus, a file copy happens instantaneously, no matter how large the file is, because all that happens is that the source file extents (references to contiguous regions of storage blocks where its data reside) are duplicated for the destination file, without actually copying the data. In addition, this scheme also conserves the space the copied file's data would require. When one of the extent-sharing files is modified, the changed blocks are copied becoming private to each file and the shared extents are disassociated—an operation known as *copy on write*.

The main Linux interface for creating such shared extents is the *copy_file_range* system call. As its name implies, it copies a specified range of bytes from one file to another. If the two files reside on the same filesystem, and the ranges are aligned with the filesystem's block size, and that filesystem supports shared file extents, then data between the two files will be shared. Even if these conditions aren't met, writing data with this system call can boost performance by allowing the OS to schedule read and write operations to happen asynchronously in a parallel manner, something that's tricky to do in a user program.

I enhanced *sed*'s I/O implementation to create shared file extents by modifying the code path that handles output (see Figure 3). At the point where data are output, I check whether the input and output file descriptors are associated with regular files, and, if so, I call a function that eventually issues the *copy_file_range* system call. Intermediate function call layers handle compatibility for operating systems and file systems that don't support this feature, block alignment (byte sequences that aren't aligned with a file's block size aren't shared), and the possibility that the call copies fewer than the requested bytes, e.g., due to an interrupt.

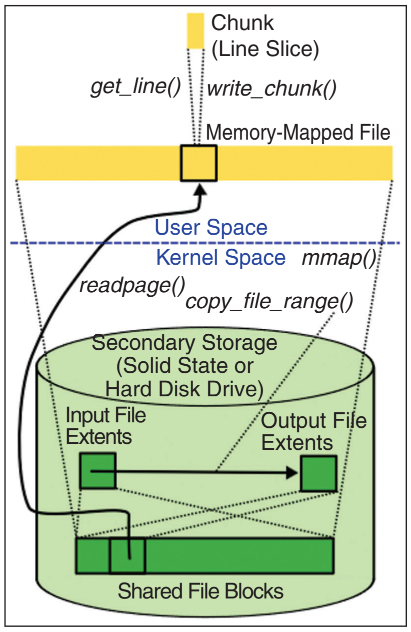


FIGURE 3. Sharing file extents.

Compared to the original *mmap()*-based I/O, the file extent optimization can be up to 2.9 times faster on solid-state disk storage; on magnetic disks, performance gains would be even more impressive. On typical workloads, I measure 3 to 59% performance gains.

File extents can also yield space savings. When the output file shares blocks with the input one, an examination of the two with the *xfs_io* command shows output like the following, in this case indicating that the first 857,199 blocks of both files share the same extents (storage blocks 11917304 to 12774503, inclusive) and only eight blocks at the end are using different storage areas.

```
$ xfs_io -c 'fiemap' input-file
input-file:
    0: [0.857199]: 11917304.12774503
    1: [857200.857207]: 12774504.12774511
$ xfs_io -c 'fiemap' output-file
output-file:
    0: [0.857199]: 11917304.12774503
    1: [857200.857207]: 13771984.13771991
```

Lessons Learned

The optimization methods I described exemplify three lessons regarding I/O performance optimization.

First, avoid the cost of copying data. Copying does not contribute to the actual work, so all eliminated copying boosts performance. As a typical example, OS network stacks often build network packets from the inside out. They leave padding around the packet's data, so that each successive layer (e.g., Ethernet, then IP, then TCP) can add there its required headers without incurring the cost of copying the data payload.

Second, utilize system support for fast I/O. Both hardware and operating systems provide diverse means for optimizing I/O. At the hardware level, this may involve DMA, direct access to persistent NVMe (Non-Volatile Memory Express) storage, and the offloading of network operations to the hardware interface. At the operating system level I/O optimizations are available through memory-mapped files, the reading of data directly into user memory (via the O_DIRECT open flag), the *copy_file_range* system call for copying between files, and the *sendfile* system call for transferring file data over a network socket.

Third and most general, separate the slow from the fast path.[5] There are often cases where some operations of the same class can be performed a lot more efficiently than others. By recognizing these and processing them separately through a fast path special case, overall performance can be boosted if these take up a significant percentage of processing time. In the case of *sed*'s I/O optimizations the fast paths included memory-mapped file I/O, output to files residing in file systems supporting shared file extents, and I/O of block-aligned data.

What is the overall gain from *sed*'s I/O optimizations? The best case involves *sed* copying a file on a filesystem supporting shared file extents, such as XFS. I measured *sed*'s performance when processing a five million-line, 419 MB file. In this case the optimized Rust *sed* implementation is

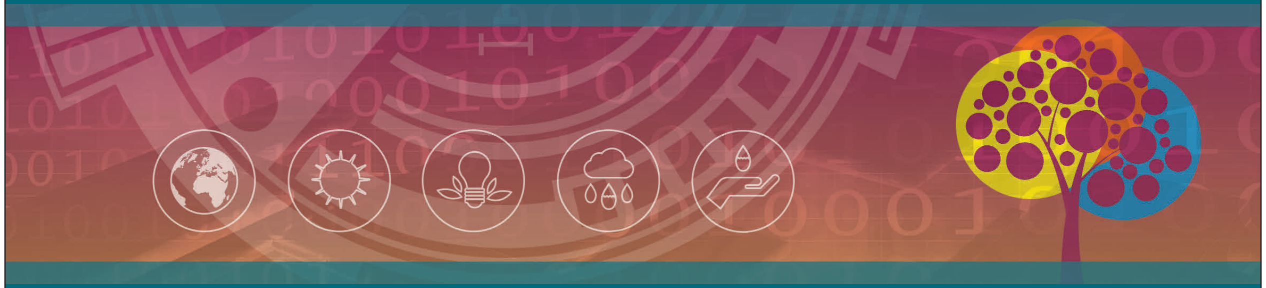
5.5 times faster than GNU sed, which is currently distributed with most Linux systems.

References

1. L. E. McMahon, “SED—A non-interactive text editor,” in *UNIX Programmer’s Manual —Supplementary Documents*, vol. 2, 7th ed. Murray Hill, NJ, USA: Bell Telephone Laboratories, 1979, pp. 1–10.
2. D. Spinellis, “Rewriting the Unix stream editor in Rust,” *IEEE Softw.*, vol. 42, no. 5, pp. 21–25, Sep./Oct. 2025, doi: 10.1109/MS.2025.3579008.
3. D. Spinellis, “Designing a programmable stream editor,” *IEEE Softw.*, vol. 42, no. 6, pp. 23–27, Nov./Dec. 2025, doi:10.1109/MS.2025.3597697.
4. D. Spinellis, “Another level of indirection,” in *Beautiful Code: Leading Programmers Explain How They Think*, A. Oram and G. Wilson, Eds., Sebastopol, CA, USA: O’Reilly and Associates, 2007, ch. 17, pp. 279–291.
5. J. H. Saltzer and M. F. Kaashoek, *Principles of Computer System Design*. Elsevier, 2009, sect. 6.1.3.

ABOUT THE AUTHOR

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology at the Athens University of Economics and Business, 104 34 Athens, Greece, and a professor of software analytics in the Department of Software Technology, Delft University of Technology, 2600 AA Delft, The Netherlands. He is a Senior Member of IEEE. Contact him at dds@aueb.gr.



Digital Object Identifier 10.1109/MS.2025.3635848