

Why stdout is faster than stderr?

=====

50 minute read Published: 2024-01-10

I recently realized stdout is much faster than stderr for Rust. Here are my findings after diving deep into this rabbit hole.



I have been using the terminal (i.e. command-line) for most of my day-to-day things for a while now. I was always fascinated by the fact that how quick and convenient the command-line might be and that's why I'm a proponent of using CLI (command-line) or TUI (terminal user interface) applications over GUI (graphical user interface) applications, when it is possible. On top of my already existing preference, I started to wholeheartedly believe that the terminal is the future after seeing the recent developments in the terminal user experience with tools like [Zellij](#) and GPU-powered terminal emulators such as [Alacritty](#) / [Wezterm](#) / [Rio](#). When this huge potential is combined with a robust systems programming language such as [Rust](#), the result is often times a very smooth terminal and development experience which I think every developer appreciates when it comes to effectiveness, speed and safety.

That is most likely why I was drawn into building terminal user interface applications with Rust in the first place. When I built my first ever Rust/TUI project, [kmon](#), I was surprised by how the limits of a simple thing such as a terminal can be pushed to build applications which gets you addicted to using terminals even more. Couple of years later, I'm one of the maintainers of [Ratatui](#) which is a Rust library for cooking up TUIs and I'm blessed to be one of the core team members which revived the unmaintained [tui-rs](#) library as Ratatui last year.

When you take all of this into account, as a daily terminal user and a command-line developer, I'm tackling new terminal related issues every day. Sometimes I come across really interesting questions and problems. As you might expect, this blog post is the fruit of one of those questions.

```
>
> Why stdout is faster than stderr?
>
```

Okay, now let's take a step back and try to understand the question first. We need to grasp some concepts about UNIX before everything.

► Table of Contents

I/O Streams

UNIX operating system brought many groundbreaking advances into the world of computers and undoubtedly one of them was the standard streams. According to UNIX, every process has three streams opened for it when it starts up:

0. Standard Input (``stdin``): for reading input.
1. Standard Output (``stdout``): for writing conventional output.
2. Standard Error (``stderr``): for printing diagnostic or error messages.

Here is an oversimplified example to demonstrate these streams:

```
# read the value of foo from stdin
$ read -r foo
test

# print the value of foo to stdout
$ echo "value of foo is '$foo'"
value of foo is 'test'

# "echoo" command does not exist so an error message will be printed to stderr
$ echoo "$foo"
bash: echoo: command not found
```

These I/O (input/output) streams are typically attached to the user's terminal via tty (TeleTYpe) which can be described as an interface that enables access to the terminal.

As you might have heard multiple times, "everything is a file" according to the UNIX philosophy. This means that the I/O streams should also be a file and this is in fact true:

```
$ file /dev/stdin /dev/stdout /dev/stderr

/dev/stdin: symbolic link to /proc/self/fd/0
/dev/stdout: symbolic link to /proc/self/fd/1
/dev/stderr: symbolic link to /proc/self/fd/2
```

If you have realized, the file descriptor (unique identifier) of these *abstract* files are the same as the initial list given above (starts from 0).

: So if they are files, we should be able to read them right?

Actually, no-

: *types quickly*

```
$ cat /dev/stdout
```

🐼: Why nothing is happening?

sigh It is because they are not real files, they are just file descriptors linked to TTY or PTY (i.e. emulated TTY AKA PseudoTeletype).

```
$ ls -l /proc/self/fd/
```

```
lrwx----- - orhun 0 0 -> /dev/pts/20
```

```
lrwx----- - orhun 0 1 -> /dev/pts/20
```

```
lrwx----- - orhun 0 2 -> /dev/pts/20
```

As you can see, the standard streams are attached to PTYs (i.e. pseudo-terminals) under `~/dev/pts`.`

🐼: Wait, so `~/dev/stdout`` is a symlink to a file descriptor at `~/proc/self/fd/1`` and that file descriptor is a symlink to `~/dev/pts/20`!`?

What even is `~/dev/pts/20`` in this case?

```
$ file /dev/pts/20
```

```
/dev/pts/20: character special (136/20)
```

🐼: bruh, what?

In Unix, character special files are files that access to I/O devices such as a NULL file (`~/dev/null``) and file descriptors. In our case, `~/stdout`` is a file descriptor (1) so it is a character special file. (The name "character special" actually comes from the fact that each character is handled individually.)

Each character special file has a device *major* number, which identifies the device type, and device *minor* number, which identifies a specific device of a given device type. So what you see on the right (136/20) means:

```
$ rg '136' /proc/devices
```

```
136 pts
```

```
- Major 136: a PTS device (~/dev/pts`)
```

```
- Minor 20: device 20 (~/dev/pts/20`)
```

🐼: Ok cool, but why am I not able to read from `~/dev/stdout``?

Oh that, yes. When you try to read data from stdout, it keeps running because it's waiting for data to read from the file descriptor. So if you give it input, then you can read it back:

```
$ cat /dev/stdout
```

```
foo # input
foo
bar #input
bar
```

Now that we have a general understanding of I/O streams, we can jump to the real world examples and make progress towards answering our question.

TUI Applications and I/O

```
>
> The following section makes use of the Rust programming language but the overall concepts are
> generally applicable to any programming language - even HolyC.
>
```

Terminal user interfaces leverages the terminal by drawing widgets/components such as text inputs, spinners and styled text on it, similar to the traditional graphical user interfaces. The terminal is able to *render* such elements thanks to the custom handling of ANSI escape codes. These ANSI *sequences* are used to control the cursor location, color and styling of the terminal.

So in order to create a terminal user interface, we need a *low-level library* for controlling both the terminal and I/O streams and also rendering the UI components. Usually, this two step process is split between different libraries for ease-of-use and the sake of single responsibility principle.

For example, while ncurses (one of the oldest TUI libraries written in C) is taking care of the low-level interface to the terminal, CDK (curses development kit) provides a set of widgets to build GUI-like applications in the terminal.

Similarly in the Rust ecosystem, the following libraries are the most preferred ones for this task today:

- crossterm : pure-Rust cross-platform terminal manipulation library.
- ratatui : lightweight library that provides a set of widgets and utilities - supports different backends including `crossterm`.

So let's build a very simple TUI using these libraries:

► [simple-tui.rs](#) (click here to expand)

You can run this using rust-script as follows:

```
$ cargo install rust-script
# [...]

$ chmod +x simple-tui.rs
```



```
    }  
  }  
}
```

3. Rendering widgets

And lastly, this is where ``ratatui`` shines:

```
use ratatui::{prelude::*, widgets::*};  
  
frame.render_widget(  
  Paragraph::new("__QQ\n(_)_\">")  
    .alignment(Alignment::Center)  
    .block(  
      Block::default()  
        .title("blog.orhun.dev")  
        .borders(Borders::ALL),  
    ),  
  frame.size(),  
);
```

Here, we are creating two widgets:

- Paragraph: contains a centered text
- Block: wraps the paragraph with a title

``ratatui`` provides many other widgets and makes it surprisingly easy to build complex interfaces. Also, as you can see from the example, it is actually easy to center a `div` text.

Going back to our original topic, if you take a look at the ``main`` function again, this all happens on `*stdout*`:

```
use std::io::stdout;  
  
let mut terminal = Terminal::new(CrosstermBackend::new(stdout()))?;
```

Obviously, we can try changing all the references of ``std::io::stdout`` to ``std::io::stderr`` and try running again.

```
-let mut terminal = Terminal::new(CrosstermBackend::new(std::io::stdout()))?;  
+let mut terminal = Terminal::new(CrosstermBackend::new(std::io::stderr()))?;
```

🐇: ... then what happens?

Well, nothing changes visually and we can't tell a difference. It is the same result.

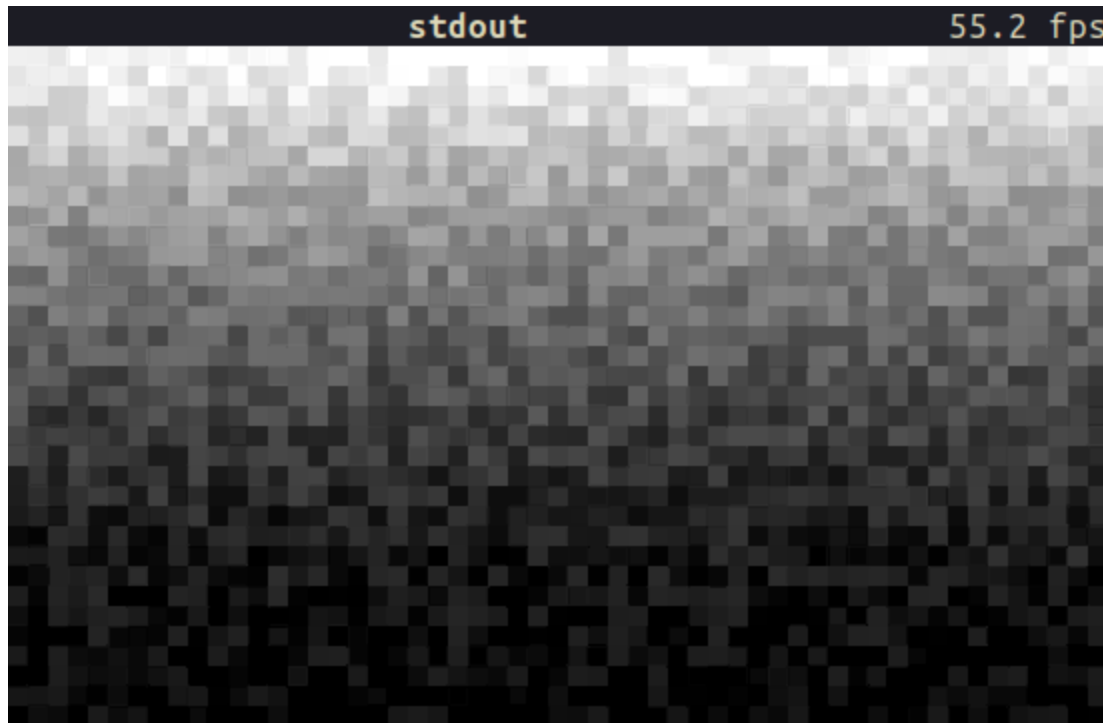
Or is it?

Measuring FPS

We need a way of measuring the performance of the rendered TUI and see the difference between using stdout and stderr. For that, I came up with a FPS (frames per second) counter application along with some monochrome colors for visualization (based on `ratatui`'s [colors_rgb](#) template).

► [stdout-vs-stderr.fps](#) (click here to expand)

Press `*space*` to switch between stdout and stderr:



Did you notice the FPS drop? stdout is ~2x faster than stderr on a 550x360 terminal!

Profiling

Before diving into the Rust code, let's take a look at the runtime externally via observing the CPU and system calls to understand what is going on.

For this task I will use a profiling tool called [samplify](#).

```
>  
> `samplify` is a command line CPU profiler which uses the Firefox Profiler as its UI.  
>
```

It records a profile of the given command's execution and then opens [profiler.firefox.com](#) in the browser where we can inspect a bunch of stuff like which functions were running for how long, flame graphs and timelines. We can even see the `*source code*` for the calls and which lines were sampled how many times.

Let's start by installing ``samplify``:

```
$ cargo install samply
```

(I recently packaged it for Arch Linux so it is also available via ``pacman -S samply`` btw)

And then we need to make some changes to the code that we are going to profile. For Rust projects, it is recommended that we build in `*release mode with debug info*` for getting inline stacks and source code view. So we can add the following profile to our ``Cargo.toml``:

```
[profile.profiling]
inherits = "release"
debug = true
```

Also, I made some changes to the previous ``stdout-vs-stderr.rs``:

- Removed the parts such as FPS widget which are irrelevant for profiling.
- Added ``STREAM`` environment variable for starting the TUI with the specified I/O stream.
 - Accepts either "stdout" or "stderr"
- Added ``DURATION`` environment variable for exiting the TUI after a certain number of seconds.

► [stdout-vs-stderr-profiler.rs](#) (click here to expand)

Now we can build the binary with the ``profiling`` profile:

```
$ cargo build --profile profiling
```

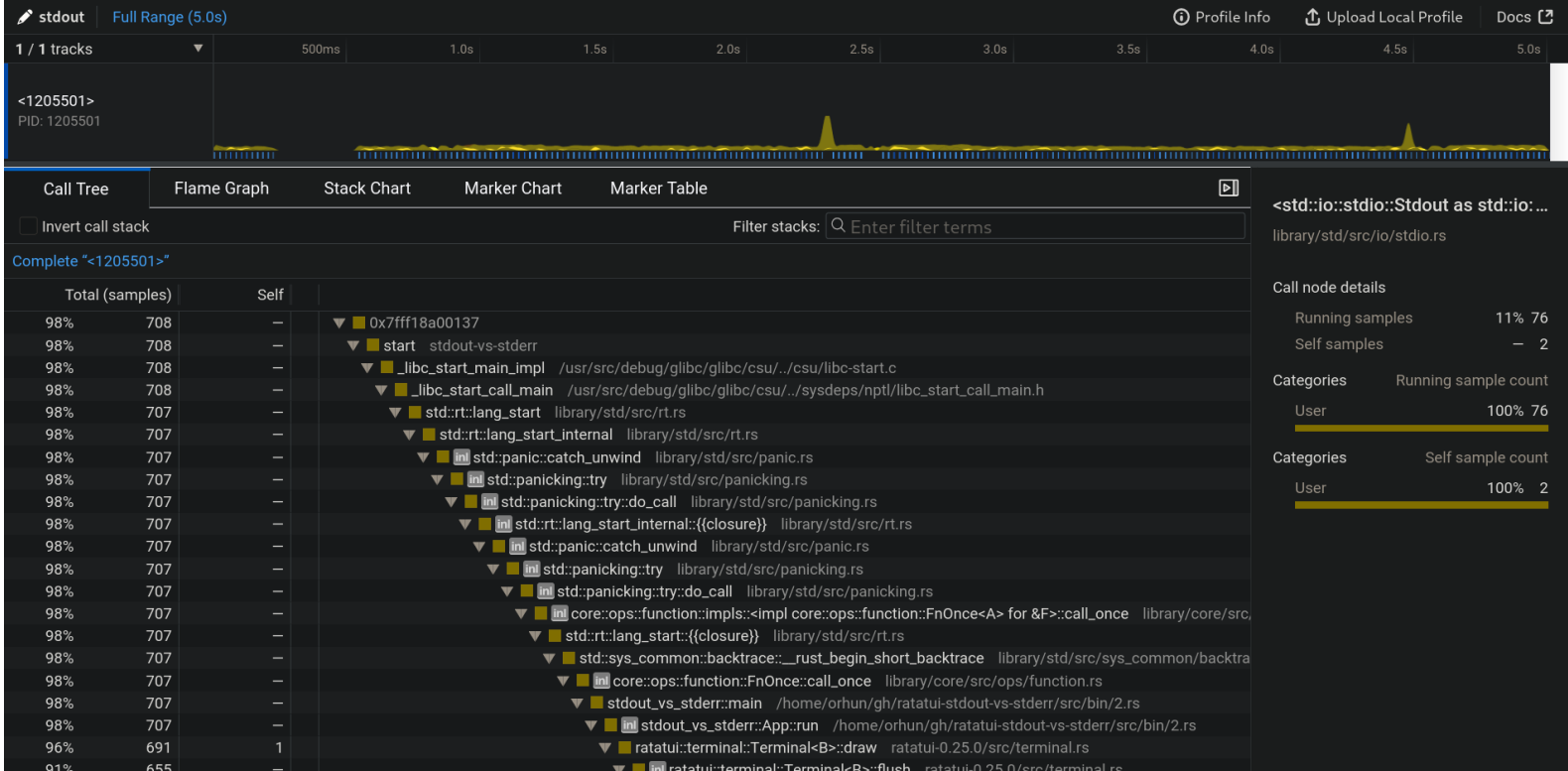
To record a profile, simply run ``samply``:

```
$ export STREAM="stdout"
$ export DURATION=5
```

```
$ samply record target/profiling/stdout-vs-stderr-profiler
```

(Or you can use my [`run-profiler.sh`](#) script which does everything for you.)

Once the TUI exits after 5 seconds, ``samply`` will greet us with `profiler.firefox.com`:



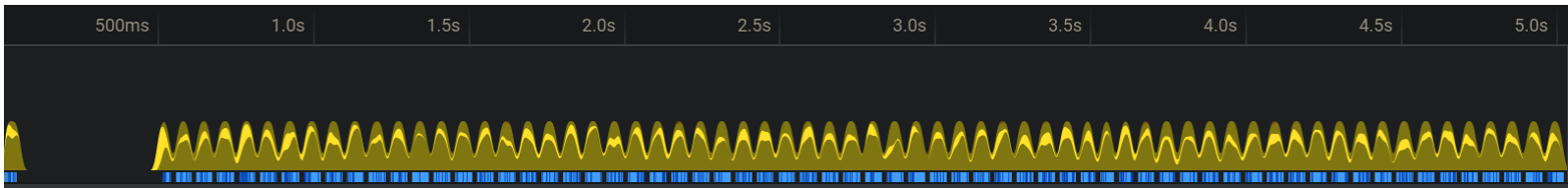
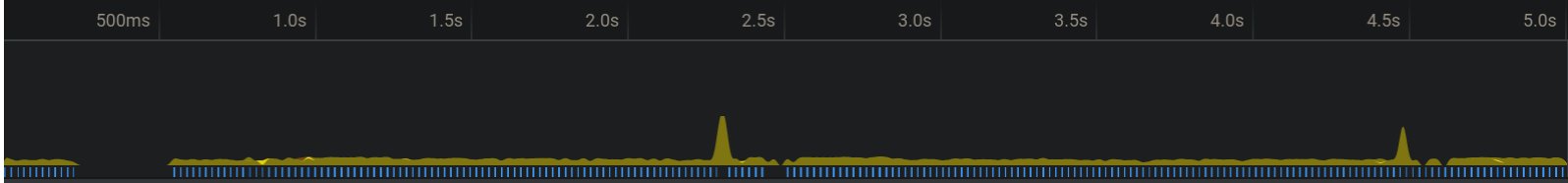
After we do the same for `stderr` via `STREAM="stderr"`, we can start comparing what went differently for the I/O streams.



These profiles are also available for online browsing if you want to experiment yourself:

- [stdout profiling](#) (download as a file)
- [stderr profiling](#) (download as a file)

Right off the bat, you can see the difference on the CPU view (first one being stdout):



stdout has 708 samples compared to stderr which has 3,315 which means stdout made 4x less calls than stderr in the course of 5 seconds! We are definitely on to something.

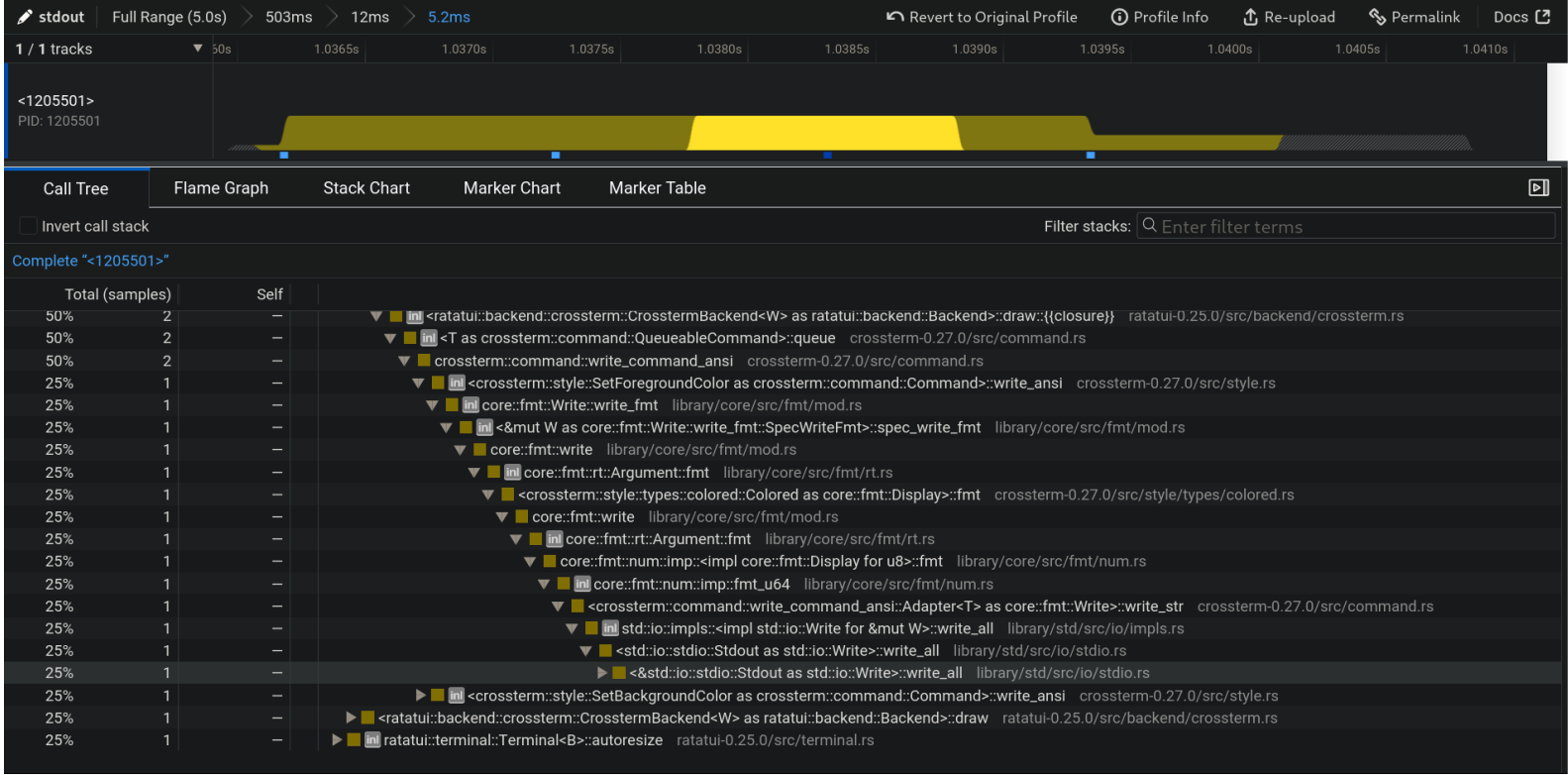
Next, we can figure out the characteristics of the each call. It is safe to assume that there will be some **write calls** to the terminal for each render. We can zoom in on the CPU view to see the each render more clearly:



As you can see, stdout rendered more frames in the same timespan as stderr. Also, there are more lighter yellow spikes happening for stderr (on every render) compared to stdout (occasionally).

We will figure out what those spikes are shortly.

We can zoom in even more and take a look at the calls that have happened for each render:



`<std::io::stdio::Stdout as std::io::Write>::write_all``

► Full Stack Trace (click here to expand)



`<std::io::stdio::StderrLock as std::io::Write>::write_all``

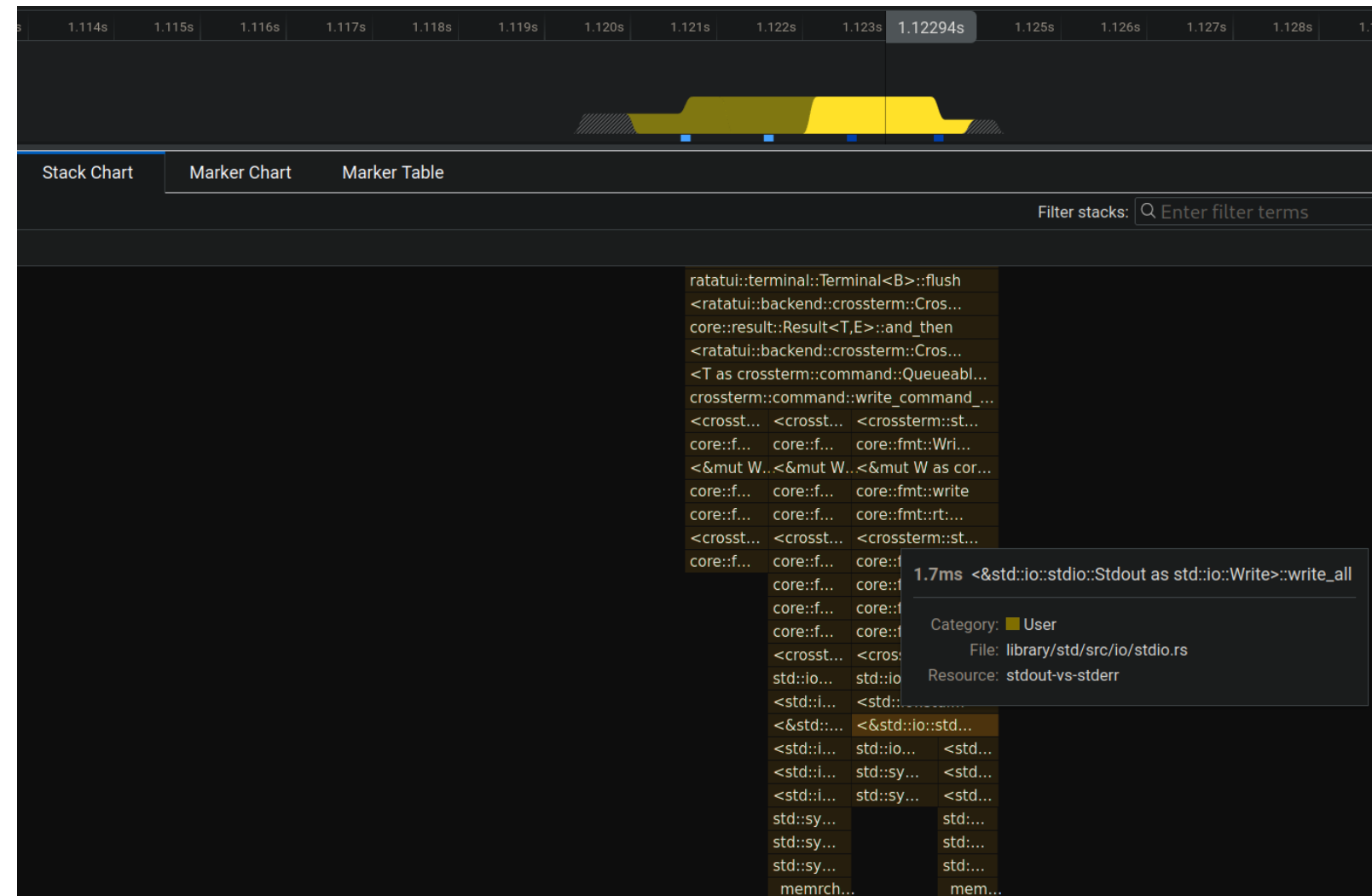
► Full Stack Trace (click here to expand)

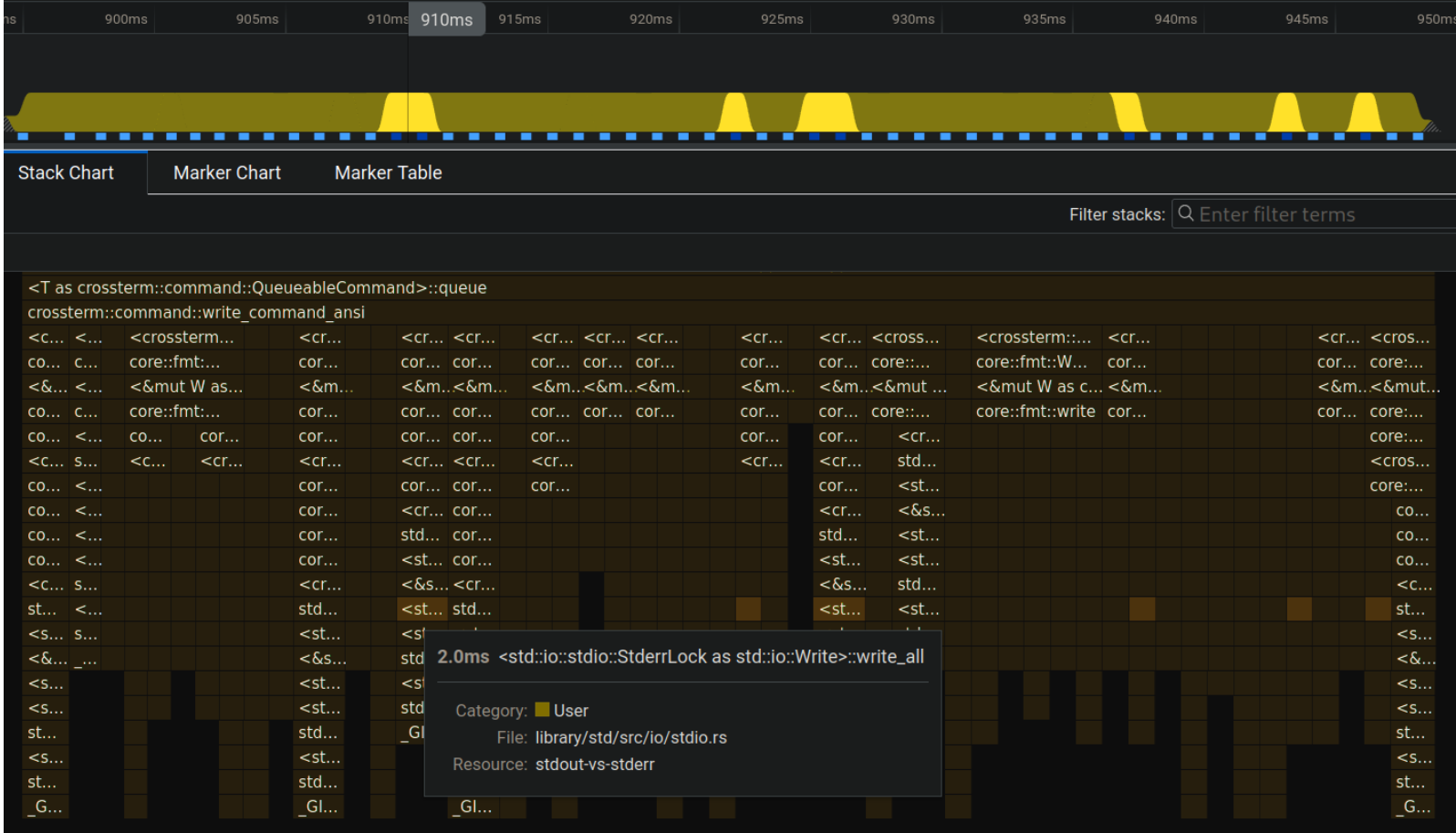
🐛: Now it makes sense, those spikes were `write_all`` calls. It is nice to find that out, but what does that even mean?

It means that:

- `stdout` called `write_all` once in 5.2ms and it happens every once in a while.
- `stderr` called `write_all` 5 times in 66ms and it is called multiple times for almost every render.

We can see this more clearly from the stack chart (first one being `stdout`):





Lastly, we can check the code of this call but it is not really helpful since it is abstracted:

```
Complete "<1205507>"
Total (samples) | Self |
100% | 2 | - | std::io::Impis::<Impis::write_for &mut w>::write_all | library/std/src/io/impis.rs
100% | 2 | - | <std::io::Stderr as std::io::Write>::write_all | library/std/src/io/stdio.rs
100% | 2 | - | <std::io::Stderr as std::io::Write>::write_all | library/std/src/io/stdio.rs
100% | 2 | - | <std::io::StderrLock as std::io::Write>::write_all | library/std/src/io/stdio.rs

library/std/src/io/stdio.rs
Total | Self
910 | }
911 | fn flush(&mut self) -> io::Result<> {
912 |     (&mut self).flush()
913 | }
914 | fn write_all(&mut self, buf: &[u8]) -> io::Result<> {
2 | 915 |     (&mut self).write_all(buf)
916 | }
917 | fn write_all_vectorized(&mut self, bufs: &mut [IoSlice<'_>]) -> io::Result<> {
918 |     (&mut self).write_all_vectorized(bufs)
```

🤖: So, what is the take away from all of this?

The conclusion is that stdout making less write calls thus able to render more frames in the same amount of time. In other words, stderr blocks until the write function for each frame is rendered to the terminal whereas stdout returns faster.

There must be some buffering happening for stdout.

Testing the buffered theory

Let's remember the change that we have made which led us to this point:

```
-let mut terminal = Terminal::new(CrosstermBackend::new(std::io::stdout()))?;
+let mut terminal = Terminal::new(CrosstermBackend::new(std::io::stderr()))?;
```

So, something **internal** changes when we do this. ``ratatui`'s CrosstermBackend`` is a good place to look first:

>

> The `CrosstermBackend`` struct is a wrapper around a writer implementing `Write``, which is used to send commands to the terminal. It provides methods for drawing content, manipulating the cursor, and clearing the terminal screen.

>

```
/// A Backend implementation that uses Crossterm to render to the terminal.
```

```
pub struct CrosstermBackend<W: Write> {
    writer: W,
}

impl<W> CrosstermBackend<W>
where
    W: Write,
{
    pub fn new(writer: W) -> CrosstermBackend<W> {
        CrosstermBackend { writer }
    }
}
```

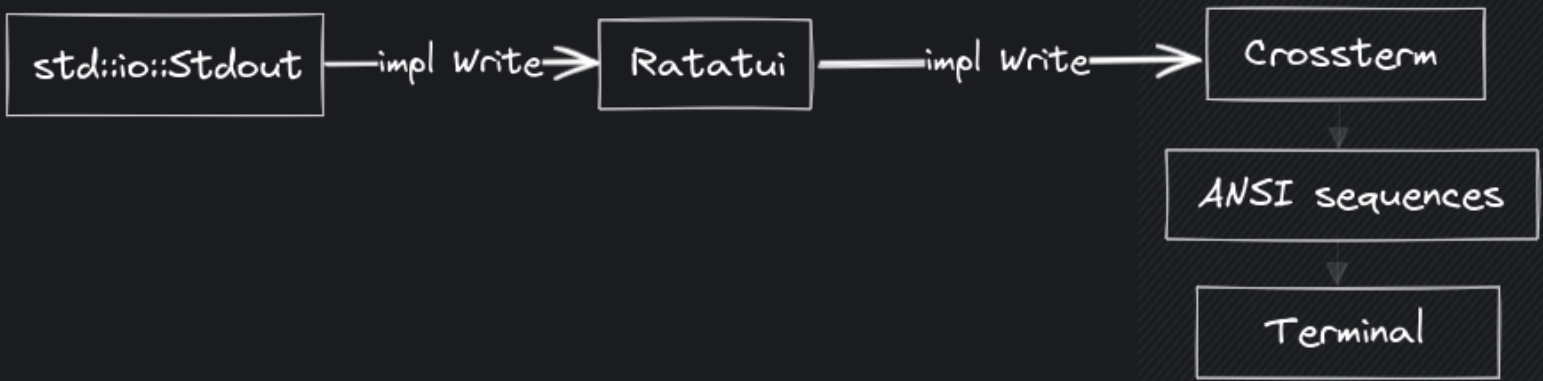
🐇: `Write``?

`Write`` is a Rust trait for objects that can be used for writing byte streams. It has methods such as `write``, `flush`` and most importantly `write_all`` which we encountered before. The code above means that `CrosstermBackend`` can work with anything that implements `Write`` such as `File``, `&mut [8]`` and other types including `Stdout``. This abstraction helps us to use the backend with different structs.

That's why the compiler is not complaining when we change the argument from `Stdout`` to `Stderr``.

🐇: Okay so `ratatui`` uses `stdout`` as a writer and sends it to `crossterm`` for writing. In this case, we need to go one layer deeper to `crossterm`` to see what it is doing different for `stdout`` and `stderr``.

Actually, no. Take a look at this diagram:



As you go deeper, the capabilities of the passed type are constrained by the implementation. In other words, if you accept a parameter that is `Write`, you have only a set of functionality you can work with (`write`, `write_all`, etc.) In this case, `Crossterm` does not have any chance to tell `stdout` from `stderr` to act different upon it, because it only knows about "a type" that is `write`.

That's why we need to go **shallower**, that being the point we started, `Stdout` and `Stderr` structs.

🐇: Wait, are you saying that we need to check out the sourcecode of the Rust standard library to get an answer to this? Doesn't that mean that `stdout` is **not always faster** than `stderr` and it depends on the implementation details?

Exactly. "Everything is a file", remember? `stdout` and `stderr` are also files which are no different. Rust must be doing some magic in this case. ✨

To find out the magic, we can take a look at the definition of `Stdout`:

```

pub struct Stdout {
    // FIXME: this should be LineWriter or BufWriter depending on the state of
    //         stdout (tty or not). Note that if this is not line buffered it
    //         should also flush-on-panic or some form of flush-on-abort.
    inner: &'static ReentrantMutex<RefCell<LineWriter<StdoutRaw>>>,
}
  
```

[std/src/io/stdio.rs#L535-L540](#)

Now let's see `Stderr`:

```

pub struct Stderr {
    inner: &'static ReentrantMutex<RefCell<StderrRaw>>,
}
  
```

[std/src/io/stdio.rs#L778-L780](#)

Take a look at the diff:

```
-ReentrantMutex<RefCell<LineWriter<StdoutRaw>>>  
+ReentrantMutex<RefCell<StderrRaw>>
```

🐞: Hmm... So `Stdout` is additionally wrapped in another struct called `LineWriter`.

Yes, do you see where this is going?

```
>  
> `LineWriter` wraps a writer and buffers output to it,  
> flushing whenever a newline (`\n`) is detected.  
>
```

Bingo!

```
>  
> We can use `LineWriter` to write one line at a time, significantly reducing the number of actual  
> writes to the file.  
>
```

This is what we were looking for all along. Let's try it!

```
use std::io::{self, LineWriter, Write};  
use std::thread;  
use std::time::Duration;  
  
let stdout = io::stdout();  
let mut writer = LineWriter::new(stdout);  
  
writer.write_all(b"In Rust's domain where choices gleam,")?;  
eprintln!("[waiting for newline]");  
thread::sleep(Duration::from_secs(1));  
  
// No bytes are written until a newline is encountered  
// (or the internal buffer is filled).  
writer.write_all(b"\n")?;  
eprintln!("\n[writing the rest]");  
thread::sleep(Duration::from_secs(1));  
  
// Write the rest.  
writer.write_all(  
    b"Ratatui's path, a unique stream.  
Terminal canvas, colors bright,  
Untraveled road, a different light.  
That choice, the difference, in code's delight.",  
)?;  
  
// The last line doesn't end in a newline,  
// so we have to flush or drop the `LineWriter` to finish writing.  
eprintln!("\n[flush or drop to finish writing]");
```



```
thread::sleep(Duration::from_secs(1));
writer.flush()?;
```

linewriter.rs

If we run this code:

```
[waiting for newline]
In Rust's domain where choices gleam,

[writing the rest]
Ratatui's path, a unique stream.
Terminal canvas, colors bright,
Untraveled road, a different light.

[flush or drop to finish writing]
That choice, the difference, in code's delight.
```

From this output, we can observe:

- First ``eprintln!`` message is printed and the writer waits for newline character to write to stdout (even though we already called ``write_all`` before).
- The second part of the poem is printed as a whole until the last line.
- Last line is not being printed until we flush the stdout.

Although this might seem like a pretty weird behavior at first glance, it actually has a huge performance benefit and it is the reason why stdout is **much faster** than stderr!

See you in another blog pos-

🙄: Wait! Is that it?

You're right, we can actually do more with this information.

```
## Experimenting with buffered writes
```

Let's go back to the standard library definition of ``Stdout``:

```
pub struct Stdout {
    // FIXME: this should be LineWriter or BufWriter depending on the state of
    //         stdout (tty or not). Note that if this is not line buffered it
    //         should also flush-on-panic or some form of flush-on-abort.
    inner: &'static ReentrantMutex<RefCell<LineWriter<StdoutRaw>>>,
}
```

🐰: Yeah, what's up with that `FIXME` comment there? Also, what is `BufWriter`?

Good questions, the documentation explains BufWriter very well:

```
>
> A `BufWriter` keeps an in-memory buffer of data and writes it to an underlying writer in large,
> infrequent batches. `BufWriter` can improve the speed of programs that make *small* and *repeated*
> write calls to the same file or network socket.
>
```

In other words, `BufWriter` writes data to its internal buffer instead of the actual stream and then infrequently writes this collected data to the stream.

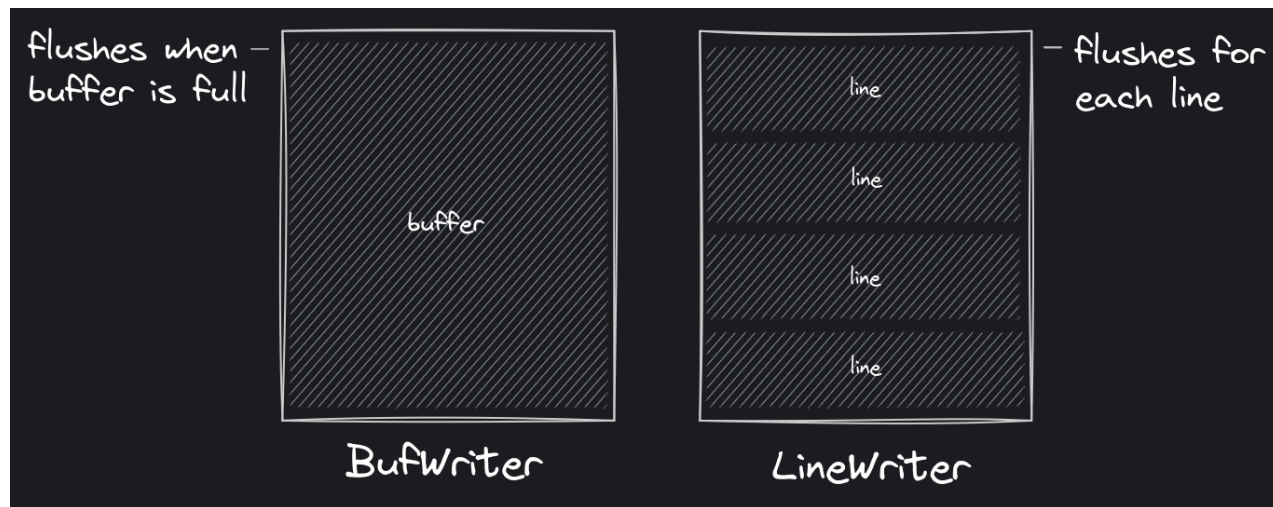
🐰: Isn't that the same thing as `LineWriter`?

Good point! They actually have a distinction when it comes to flushing (i.e. when the buffered data is written to the stream).

- `BufWriter`: flushes when the internal buffer is full.
- `LineWriter`: same behavior as `BufWriter` but also flushes for each line (when `0x0a` or `\n` is detected).

Also, they both flush when the writer goes out of scope.

To make it more clear:



Let's change our previous `LineWriter` example to use `BufWriter`:

```
use std::io::{self, BufWriter, Write};
use std::thread;
use std::time::Duration;
```

```
let stdout = io::stdout();
let mut writer = BufWriter::new(stdout);
```

```
writer.write_all(b"In Rust's domain where choices gleam,");
eprintln!("[writing the first line]");
```

```

thread::sleep(Duration::from_secs(1));

// No bytes are written until a newline is encountered
// (or the internal buffer is filled).
writer.write_all(b"\n"?;
eprintln!("\n[writing the rest]");
thread::sleep(Duration::from_secs(1));

// Write the rest.
writer.write_all(
    b"Ratatui's path, a unique stream.
Terminal canvas, colors bright,
Untraveled road, a different light.
That choice, the difference, in code's delight.",
)?;

// The last line doesn't end in a newline,
// so we have to flush or drop the `LineWriter` to finish writing.
eprintln!("\n[flush or drop to finish writing]");
thread::sleep(Duration::from_secs(1));
writer.flush()?;

```

[bufwriter.rs](https://github.com/mafintosh/bufwriter.rs)

We will see that nothing is printed until we flush the `BufWriter`:

```
[writing the first line]
```

```
[writing the rest]
```

```
[flush or drop to finish writing]
```

```
In Rust's domain where choices gleam,
Ratatui's path, a unique stream.
Terminal canvas, colors bright,
Untraveled road, a different light.
That choice, the difference, in code's delight.
```

When it comes to the comment on the stdout struct:

```

>
> This should be LineWriter or BufWriter depending on the state of stdout (tty or not).
>

```

What is being said here is that stdout should automatically decide between line buffering (`LineWriter`) and block buffering (`BufWriter`) based on whether it is attached to a TTY or not.

So stdout should no longer be *line buffered* when outputting to a non-terminal (like piping output to a file).

🐛: I didn't get it, what is the advantage of using `BufWriter`` in this case?

In the case of a non-TTY, let's say writing to a file, this would mean that the output will be **block buffered** thus we won't be making system calls for each line. In other words, we won't be flushing continuously which is a huge performance benefit. This can save us from substantial overhead when working with larger files.

And this is actually implemented in Rust but the pull request is not merged: [#115652](#)

Also, it would be super nice have a way to opt-in for block buffering for stdout in the future. There is a related discussion here: [#60673](#)

🐛: What would be the benefit of that?

Check out this code for example:

```
for i in 1..1000000 {
    println!("{}", i);
}
```

Keep in mind that `println!` is writing to stdout and it is **line buffered** (i.e. uses `LineWriter``) as default. In other words, it flushes the terminal for each line and performs a system call!

Now take a look at this:

```
let stdout = io::stdout();
let mut output = BufWriter::new(stdout);
for i in 1..1000000 {
    writeln!(output, "{}", i)?;
}
```

Here, we are wrapping stdout in a `BufWriter`` which makes it **block buffered** ✨

► [block-buffered-stdout.rs](#) (click here to expand)

When we run it:

```
$ chmod +x block-buffered-stdout.rs

$ ./block-buffered-stdout.rs

# [...]
Line buffered: 1.080789949s
Block buffered: 408.105636ms
```

Block buffered stdout runs ~2x faster!

🐇: Nice! I wonder what happens if we apply this to our TUI app.

We can try by doing this change:

```
-let mut terminal = Terminal::new(CrosstermBackend::new(stdout()))?;  
+let mut terminal = Terminal::new(CrosstermBackend::new(BufWriter::new(stdout())))?;
```

stdout (line buffered)

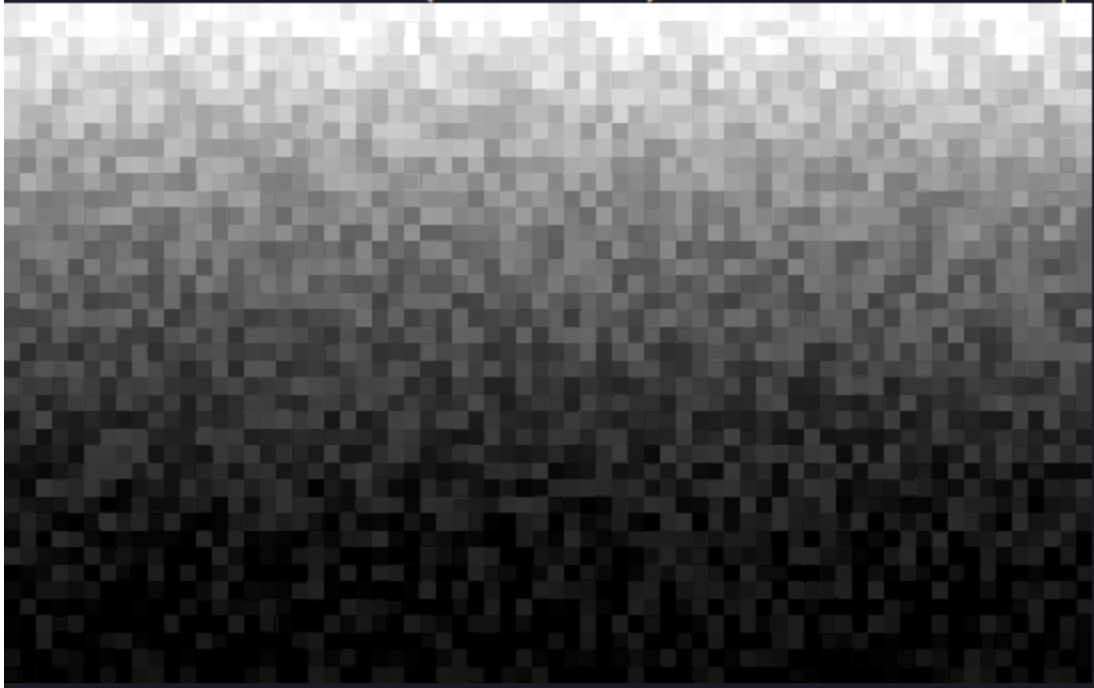
53.1 fps

Hmm, there isn't a noticeable increase in performance. What if we try something more substantial like using a block buffered stderr. The default stderr is not buffered, remember?

🐇: Yeah... Oh! I have a better idea. How about making the stderr line buffered? Stdout is also line buffered so are we going to get the same performance?

Yes, let's try that!

```
// line buffered stdout (as default)  
let mut terminal = Terminal::new(CrosstermBackend::new(stdout()))?;  
  
// line buffered stderr  
let mut terminal = Terminal::new(CrosstermBackend::new(LineWriter::new(stderr())))?;
```



🐰: Damn, did we just make the performance of stderr identical to stdout just by making it line buffered?

Oh yeah, looks like it!

Experimenting with raw writes

How about we do the opposite of what we have done so far and try to make stdout *unbuffered*. This would degrade the performance and we should probably get a result similar to using the default stderr. Let's prove our hypothesis!

If we take a look at our findings so far:

I/O Stream	Buffering
<code>`std::io::stderr()`</code>	Unbuffered/raw
<code>`LineWriter<std::io::stderr(>`</code>	Line buffered
<code>`BufWriter<std::io::stderr(>`</code>	Block buffered
<code>`std::io::stdout()`</code>	Line buffered
???	Unbuffered/raw

Since ``stderr()`` returns a raw stream as default (i.e. ``StderrRaw``), it is easier to implement a buffering layer on top of it. However, ``stdout()`` function already returns a buffered stream so we need to somehow get a *raw* stream.

If you remember the contents of ``Stdout``:

```
/// A handle to the global standard output stream of the current process.
```

```
pub struct Stdout {  
    inner: &'static ReentrantMutex<RefCell<LineWriter<StdoutRaw>>>,  
}
```

[std/src/io/stdio.rs#L535-L540](#)

We need the `StdoutRaw` for an unbuffered stream rather than having it wrapped inside `LineWriter`. The type definition also confirms the unbuffered behavior:

```
/// A handle to a raw instance of the standard output stream of this process.  
///  
/// This handle is not synchronized or buffered in any fashion. Constructed via  
/// the std::io::stdio::stdout_raw function.  
struct StdoutRaw(stdio::Stdout);
```

[std/src/io/stdio.rs#L45-L49](#)

Nice. This comment leads us to `stdout_raw` function:

```
/// Constructs a new raw handle to the standard output stream of this process.  
///  
/// The returned handle has no external synchronization or buffering layered on  
/// top.  
const fn stdout_raw() -> StdoutRaw {  
    StdoutRaw(stdio::Stdout::new())  
}
```

[std/src/io/stdio.rs#L69-L81](#)

🐞: Easy, we can just construct a raw stdout via calling `stdout_raw`!

Not really, that is a private function.

```
std::io::stdio::stdout_raw();  
    ^^^^^^ ----- function `stdout_raw` is not publicly re-exported  
    |  
    private module
```

There is actually a tracking issue from 2019 about exposing raw stdout/stderr/stdin: [#58326](#)

>

> Currently there is not easy/obvious way to get an unbuffered Stdout/err/in. The types do exist in
> stdio, however they are not public for reasons not noted. For example these types would be useful
> for CLI applications that write a lot of data at once without it getting unnecessarily flushed.

>

And sadly, there isn't still an easy/obvious way to get an unbuffered I/O streams as of now :(

But!

This issue gives us some hints about possible workarounds. One thing that is reiterated a couple of times in the issue is that we can use `from_raw_fd` on Linux as a workaround.

🐛: Let me guess, `from_raw_fd` takes a file descriptor and we are simply going to use the file descriptor of stdout (which is "1") to create an unbuffered stream.

Exactly!

```
use std::fs::File;

let mut raw_stdout = File::from_raw_fd(1);
writeln!(raw_stdout, "test");
```

But...

```
error[E0133]: call to unsafe function is unsafe and requires unsafe function or block
  |
55 |         let raw_stdout = File::from_raw_fd(1);
  |                                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ call to unsafe function
  |
  = note: consult the function's documentation for information on how to avoid undefined behavior
```

If we read the documentation of `from_raw_fd`:

```
>
> Safety: The `fd` passed in must be an owned file descriptor; in particular, it must be open.
>
```

There are more details to it (which I will cover in another blog post) but the moral of the story is, we need to put our code in a `unsafe` block like so:

```
use std::fs::File;

// SAFETY: no other functions should call `from_raw_fd`, so there
// is only one owner for the file descriptor.
let raw_stdout = unsafe { File::from_raw_fd(1) };
writeln!(raw_stdout, "test");
```

If you run it, you will see "test" on stdout. Yay! \o/

However, as mentioned briefly in the previous section, there is still a big problem with this code. Going back to the documentation of `from_raw_fd`:

> This function is typically used to **consume ownership** of the specified file descriptor. When used in this way, the returned object will take responsibility for **closing it** when the object goes out of scope.

What this means is that `raw_stdout` variable takes ownership of the file descriptor and it will *close* the stdout when it goes out of scope. In other words, when the created `File` is dropped, stdout is closed.

🐰: RIP.

We can confirm this behavior with this code:

```
use std::fs::File;
use std::io::{Result, Write};
use std::os::fd::FromRawFd;

fn print1() -> Result<()> {
    let mut raw_stdout = unsafe { File::from_raw_fd(1) };
    writeln!(raw_stdout, "test1")
}

fn print2() -> Result<()> {
    let mut raw_stdout = unsafe { File::from_raw_fd(1) };
    writeln!(raw_stdout, "test2")
}

fn main() -> Result<()> {
    print1()?;
    print2()?;
    Ok(())
}
```

raw_stdout_broken.rs

What you expect to see is "test1" and "test2", however, stdout is closed after we leave the first function. When we try to open it again, it will panic because of the safety rule (the `fd` passed in must be an owned file descriptor + it must be **open**).

```
$ ./raw-stdout-broken.rs
```

```
test1
Error: Os { code: 9, kind: Uncategorized, message: "Bad file descriptor" }
```

🐰: That's bad. What do we do?

In our case, we want the open file to *live through* the entire program. Let's also assume that this is a TUI program and we have separate functions where passing the ``raw_stdout`` value around isn't possible.

Well, there is still one quick dirty workaround: lazily initialize stdout and make it globally available via ``lazy_static`` (or another crate such as ``once_cell``):

```
use std::fs::File;
use std::io::{Result, Write};
use std::os::fd::FromRawFd;
use std::sync::Mutex;

lazy_static! {
    static ref RAW_STDOUT: Mutex<File> = unsafe { Mutex::new(File::from_raw_fd(1)) };
}

fn print1() -> Result<()> {
    writeln!(RAW_STDOUT.lock().unwrap(), "test1")
}

fn print2() -> Result<()> {
    writeln!(RAW_STDOUT.lock().unwrap(), "test2")
}

fn main() -> Result<()> {
    print1()?;
    print2()?;
    Ok(())
}
```

`raw_stdout-1.rs`

```
$ ./raw-stdout-1.rs
```

```
test1
test2
```

⚠: Okay okay, this is a bit too much. I mean lazy, static, mutex, locking etc... Don't we have a better way to handle this? Besides, you can't really construct any other stdout instances with this code.

Actually there is a better way. The documentation of ``FromRawFd`` gives us a hint:

```
>
> Consuming ownership is not strictly required. Use a `From<OwnedFd>::from` implementation for an API
> which strictly consumes ownership.
>
```

Sounds like we can work around closing the stdout if we use `OwnedFd`.

>
> An owned file descriptor. This closes the file descriptor on drop. It is guaranteed that nobody else
> will close the file descriptor.
>

So we can create a ✨ global unbuffered stdout that does not consume the ownership of the underlying file descriptor ✨ like so:

```
use lazy_static::lazy_static;
use std::fs::File;
use std::io::{Result, Write};
use std::os::fd::{FromRawFd, OwnedFd};

lazy_static! {
    static ref RAW_STDOUT_FD: OwnedFd = unsafe { OwnedFd::from_raw_fd(1) };
}

fn print1() -> Result<()> {
    let mut raw_stdout = File::from(RAW_STDOUT_FD.try_clone()?);
    writeln!(raw_stdout, "test1")
}

fn print2() -> Result<()> {
    let mut raw_stdout = File::from(RAW_STDOUT_FD.try_clone()?);
    writeln!(raw_stdout, "test2")
}

fn main() -> Result<()> {
    print1()?;
    print2()?;
    Ok(())
}
```

raw_stdout-2.rs

```
$ ./raw-stdout-2.rs
```

```
test1
test2
```

If we want a more elegant solution, we can go for `as_raw_fd` function of `Stdout` instead of using plain "1":

```
static ref RAW_STDOUT_FD: OwnedFd = {
    let stdout = std::io::stdout();
    let raw_fd = stdout.as_raw_fd();
}
```

```
unsafe { OwnedFd::from_raw_fd(raw_fd) }  
};
```

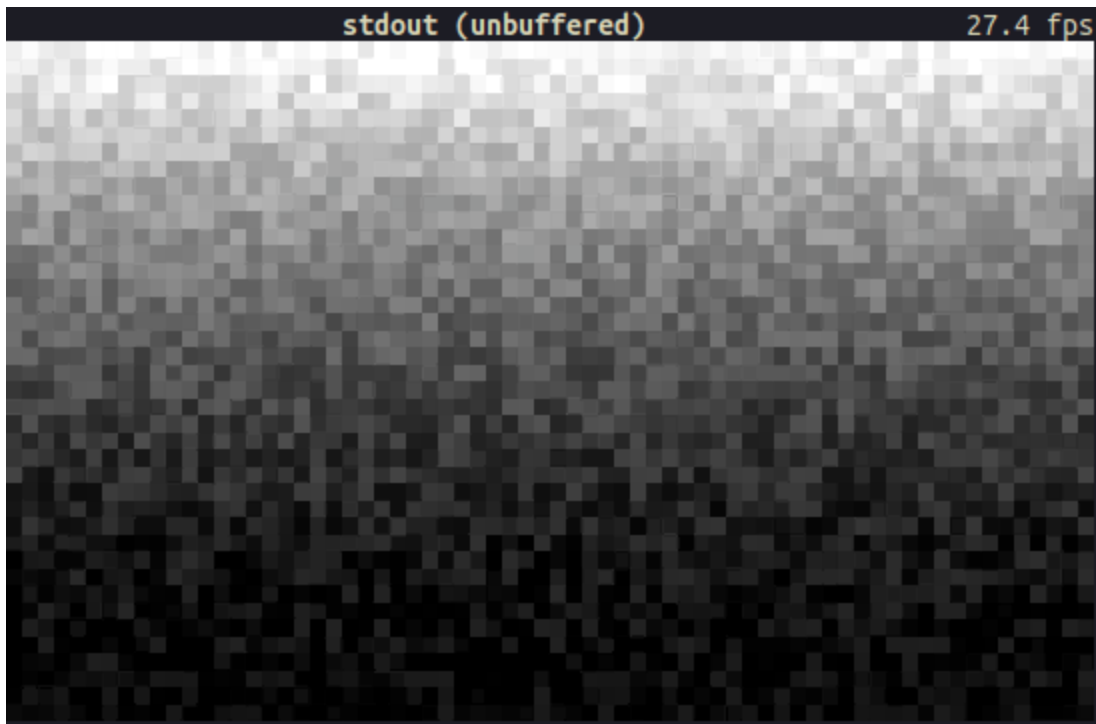
🐇: All of this pain, why?

So that we can do this:

```
let stdout = std::io::stdout();  
let raw_fd = stdout.as_raw_fd();  
let raw_stdout = unsafe { File::from_raw_fd(raw_fd) };  
  
// initialize the terminal with raw/unbuffered stdout  
let mut terminal = Terminal::new(CrosstermBackend::new(BufWriter::new(raw_stdout)))?;
```

🐇: Yeah, I almost forgot we were doing TUI stuff. I think the original question was "Is raw stdout as slow as raw stderr?".

Yes, let's find that out:



And that concludes it, raw stdout has the *same* performance as raw stderr.

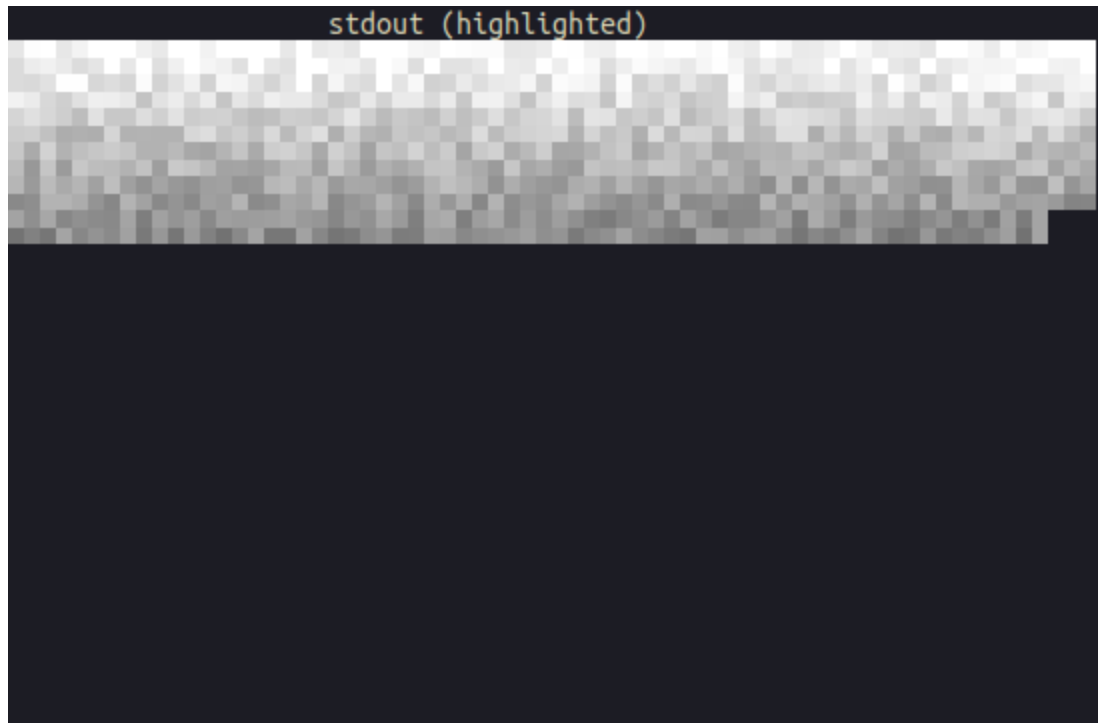
Making stdout faster

All of the things we have done so far begs the question: can we make stdout faster?

Well, we now know that the reason why stderr is slower than stdout is that it is not *buffered*. So can we somehow achieve a faster (more performant) I/O with stdout by doing something like "better buffering"?

One other thing we can do to achieve a better FPS is that reducing the `write` calls that are made. However, `Crossterm / Ratatui` is already doing some optimizations such as not rendering the cells that are not changed. Our issue is that the FPS counter example we are using has cells that are always changing so this optimization has no effect. On top of that, we set both background and foreground colors so each render essentially takes multiple `write` calls.

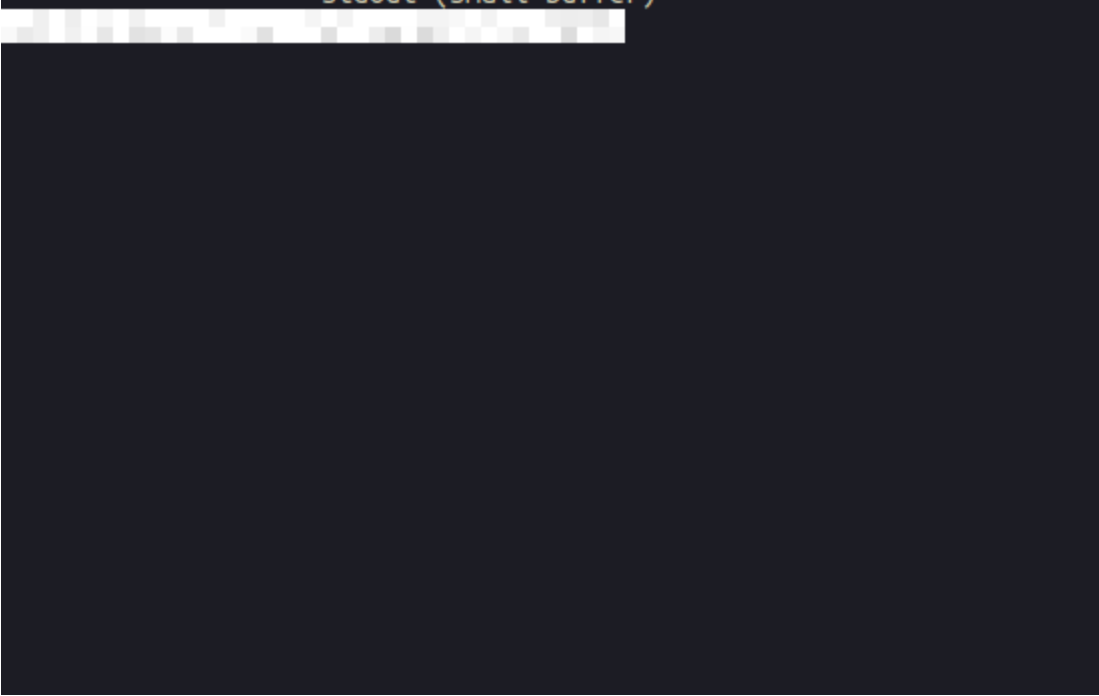
In the screencast below, I modified the `Crossterm` backend of `Ratatui` to highlight the cells that are not being changed between renders. You can clearly see from the number of red cells that we are not able skip a lot of `write` calls since mostly everything is changing on the terminal:



However this is not the case for most of the TUI applications and this optimization actually saves us from re-rendering the majority of the screen.

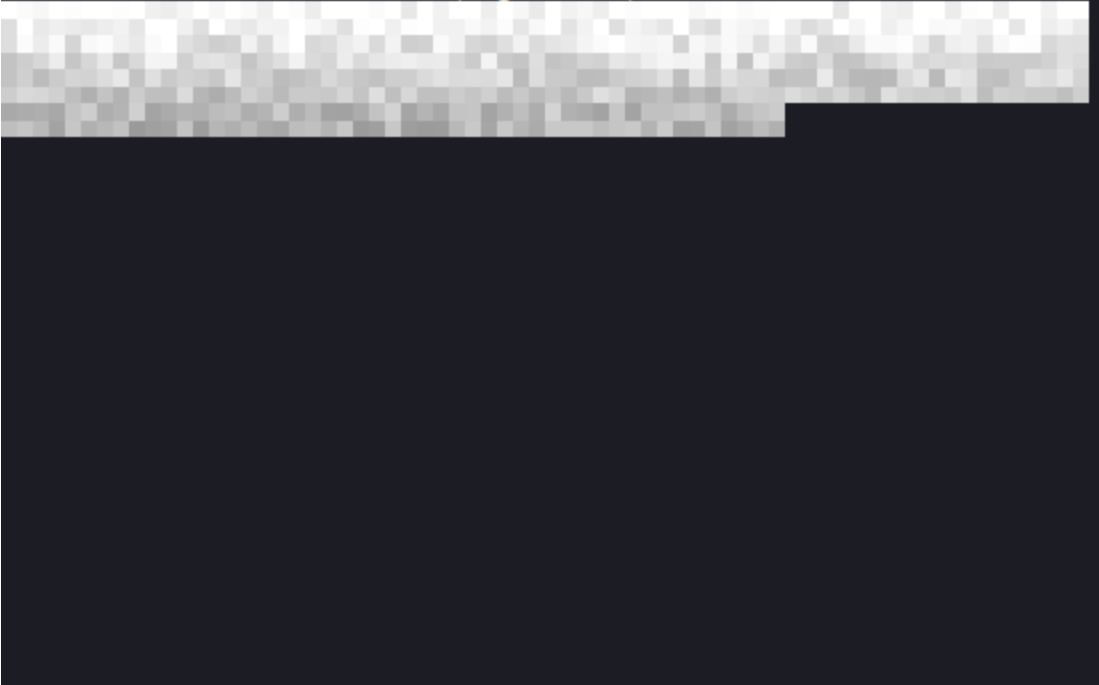
Another interesting thing to look at is the buffer size. We can observe the following with a smaller buffer size (100 bytes) and delay between renders:

stdout (small buffer)



Whereas a bigger buffer renders bigger chunks:

stdout (big buffer)



We won't see a big difference if we remove the sleep except if we use very small/big buffer then the FPS drops vastly. We can probably experiment with the buffer size to draw a single line for each render but I wasn't able to get a better FPS in my attempts.

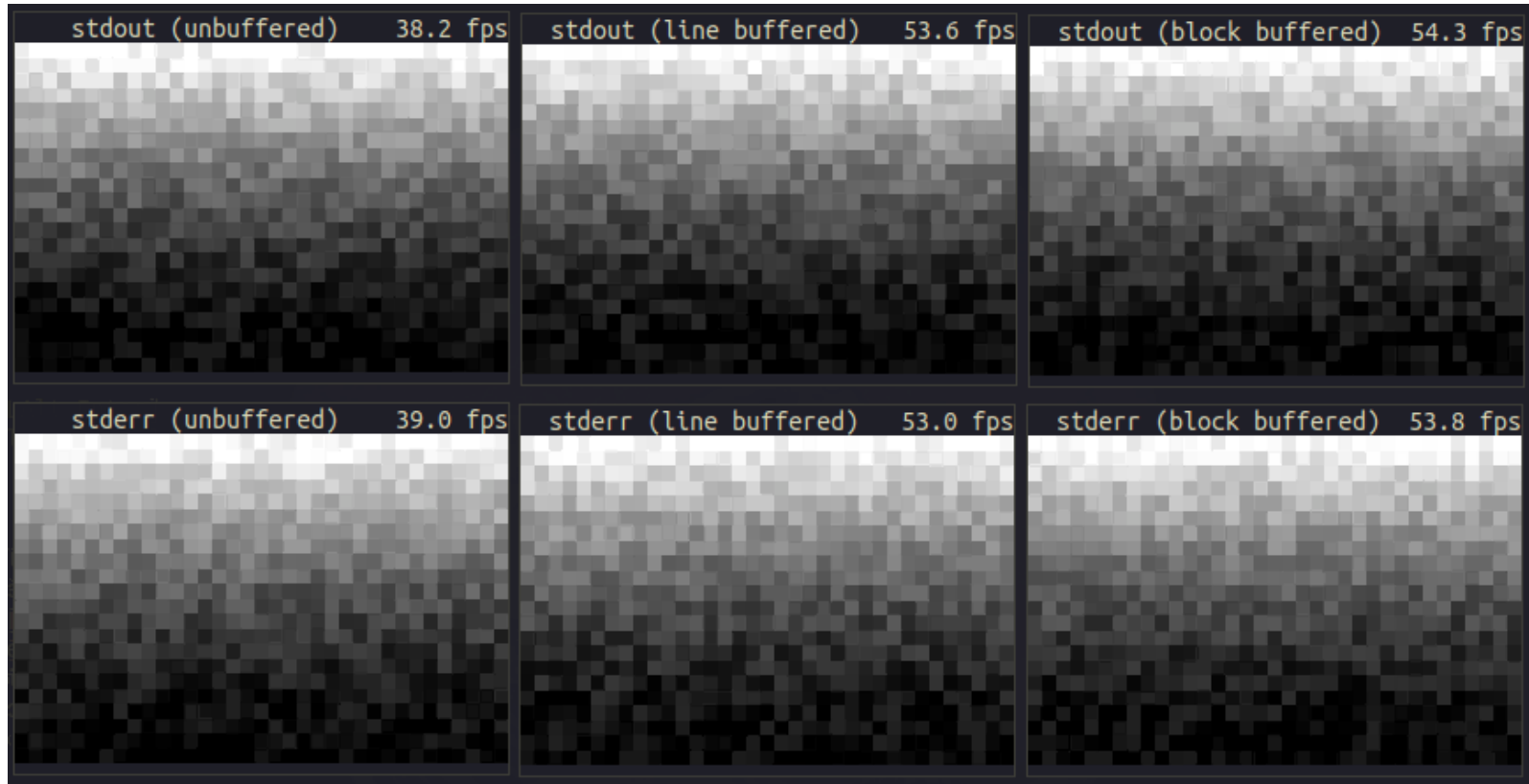
One other thing to mention, there were recent developments on ``ratatui`` to achieve a better performance for rendering cells. However, this doesn't have a big impact on FPS but definitely an improvement for using less resources.

We can keep experimenting with the low level functions of ``crossterm`` / ``ratatui`` to further optimize things but I feel like that would be a better topic for the **part 2** of this post.

While writing this, I wasn't able to achieve a "faster stdout" so feel free to leave comments about your suggestions!

Findings

Here is the `ratatui` + `crossterm` rendering comparison for stdout and stderr using unbuffered / line-buffered / block-buffered writes:



[stdout-vs-stderr-all.rs](#)

The takeaway from this is that I/O streams have similar performances when the same buffering technique is used. We can also say that `std::io::stdout()` is faster than `std::io::stderr()` because the use of line-buffered vs no buffering.

If you want to reproduce the same results, I used the following environment in my experiments:

- Backend: [crossterm](#)
- TUI/rendering: [ratatui](#)
- Terminal: [alacritty](#)
- CPU: AMD Ryzen 7 4700U with Radeon Graphics (8) @ 2.000GHZ
- GPU: AMD ATI Radeon RX Vega 6
- RAM: 24GB

I'm curious about how the results might change in different systems/terminals so feel free to share your findings below!

Other Languages

What we have covered so far only applies to Rust so it would be interesting to take a look at what other programming languages are doing in terms of buffered I/O.

Go

`os.Stdout` is unbuffered as default. It is possible to make it buffered as follows:

```
f := bufio.NewWriter(os.Stdout)
defer f.Flush()
```

I'm guessing same applies for `os.Stderr` as well.

Python

When printing interactively, `sys.stdout` is line buffered. Otherwise, it is block buffered like regular text files. The `sys.stderr` is line buffered in both cases.

One cool thing about Python is that we can actually make both streams unbuffered by passing the `-u` option or setting the PYTHONUNBUFFERED environment variable.

C

>

> Now brace yourself because this might come as a bit of a surprise to you: when you `printf()` or
> `fprintf()` or use any I/O functions like that, it does not normally work immediately. For the sake
> of efficiency, and to irritate you, the I/O on a `FILE*` stream is buffered away safely until
> certain conditions are met, and only then is the actual I/O performed.
>

As you can already tell, the I/O streams are buffered as default. However, this is configurable via `setbuf()` or `setvbuf()` function:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

For example, to turn off buffering for stdout:

```
// _IONBF: stream will be unbuffered.
setvbuf(stdout, NULL, _IONBF, 0);
```

This documentation has a good explanation of the different buffering options and related examples.

Here is an example for line buffering:

```
FILE *fp;
char lineBuf[1024];
```



```

fp = fopen("somefile.txt", "r");
setvbuf(fp, lineBuf, _IOLBF, 1024); // set to line buffering
// ...
fclose(fp);

fp = fopen("another.dat", "rb");
setbuf(fp, NULL); // set to unbuffered
// ...
fclose(fp);

```

Zig

The I/O streams are unbuffered. We can achieve buffered writes as follows:

```

const std = @import("std");

pub fn main() !void {
    const out = std.io.getStdOut();
    var buf = std.io.bufferedWriter(out.writer());

    // Get the Writer interface from BufferedWriter
    var w = buf.writer();

    try w.print("check out my Zig Bits series as well!", .{});

    // Don't forget to flush!
    try buf.flush();
}

```

C++

``std::cout`` is buffered according to [this article](#) .

Couldn't find your favorite programming language on this list? Have something to add/fix?
[Feel free to contribute to this blog](#) ! <

Conclusion

We have covered:

- How I/O streams work on Unix-like systems
- How terminal user interfaces work
- How to build terminal user interfaces using Rust (w/ ``ratatui``/``crossterm``)
- How to profile applications and observing system calls (w/ ``samply``)
- Buffering for I/O streams
 - Line buffering (``LineWriter``)

- Block buffering (``BufWriter``)
- No buffering (``from_raw_fd``)
- Rust compared to other programming languages
- And other things that I probably forgot to put here!

📁 The code snippets used in this post are available [in this repository](#) .

🎉 It was a wild ride and I learned a lot! I'm hoping that I was able to share at least one thing you didn't know before.

😬 I might have skipped something very important or shared incomplete information. Feel free to comment below or [contribute](#) your edits case you have caught something. This is the beauty of these blog posts!

👏 I would like to give special thanks to the fellow ``ratatui`` maintainer Dheepak Krishnamurthy ([@kdheepak](#)) for pioneering the first steps of this research [in this discussion](#) and took time to investigate this further with me! *Kudos*!

Hope you enjoyed!

ciao!

💖 Liked this article? Want to sponsor my blog posts and have early access? Want to add your name/logo or company's badge here? Check out my [GitHub sponsorship tiers](#) !

Published by Orhun Parmaksız in [Rust](#)

🤍 Sponsor

☕ *Buy me a coffee*
