Skip to main content

Why pipes sometimes get "stuck": buffering

• terminal •

November 29, 2024

Here's a niche terminal problem that has bothered me for years but that I never really understood until a few weeks ago. Let's say you're running this command to watch for some specific output in a log file:

```
tail -f /some/log/file | grep thing1 | grep thing2
```

If log lines are being added to the file relatively slowly, the result I'd see is… nothing! It doesn't matter if there were matches in the log file or not, there just wouldn't be any output.

I internalized this as "uh, I guess pipes just get stuck sometimes and don't show me the output, that's weird", and I'd handle it by just running `grep thing1 /some/log/file | grep thing2` instead, which would work.

So as I've been doing a terminal deep dive over the last few months I was really excited to finally learn exactly why this happens.

why this happens: buffering

The reason why "pipes get stuck" sometimes is that it's VERY common for programs to buffer their output before writing it to a pipe or file. So the pipe is working fine, the problem is that the program never even wrote the data to the pipe!

This is for performance reasons: writing all output immediately as soon as you can uses more system calls, so it's more efficient to save up data until you have 8KB or so of data to write (or until the program exits) and THEN write it to the pipe.

In this example:

```
tail -f /some/log/file | grep thing1 | grep thing2
```

the problem is that `grep thing1` is saving up all of its matches until it has 8KB of data to write, which might literally never happen.

programs don't buffer when writing to a terminal

Part of why I found this so disorienting is that `tail -f file | grep thing` will work totally fine, but then when you add the second `grep`, it stops working!! The reason for this is that the way `grep` handles buffering depends on whether it's writing to a terminal or not.

Here's how `grep` (and many other programs) decides to buffer its output:

- Check if stdout is a terminal or not using the `isatty` function
  - If it's a terminal, use line buffering (print every line immediately as soon as you have it)
  - Otherwise, use "block buffering" – only print data if you have at least 8KB or so of data to print

So if `grep` is writing directly to your terminal then you'll see the line as soon as it's printed, but if it's writing to a pipe, you won't.

Of course the buffer size isn't always 8KB for every program, it depends on the implementation. For `grep` the buffering is handled by libc, and libc's buffer size is defined in the `BUFSIZ` variable. Here's where that's defined in glibc.

(as an aside: "programs do not use 8KB output buffers when writing to a terminal" isn't, like, a law of terminal physics, a program COULD use an 8KB buffer when writing output to a terminal if it wanted, it would just be extremely weird if it did that, I can't think of any program that behaves that way)

commands that buffer & commands that don't

One annoying thing about this buffering behaviour is that you kind of need to remember which commands buffer their output when writing to a pipe.

Some commands that **don't** buffer their output:

- tail
- cat
- tee

I think almost everything else will buffer output, especially if it's a command where you're likely to be using it for batch processing. Here's a list of some common commands that buffer their output when writing to a pipe, along with the flag that disables block buffering.

- grep (`--line-buffered`)
- sed (`-u`)
- awk (there's a `fflush()` function)
- tcpdump (`-l`)
- jq (`-u`)
- tr (`-u`)
- cut (can't disable buffering)

Those are all the ones I can think of, lots of unix commands (like `sort`) may or may not buffer their output but it doesn't matter because `sort` can't do anything until it finishes receiving input anyway.

Also I did my best to test both the Mac OS and GNU versions of these but there are a lot of variations and I might have made some mistakes.

programming languages where the default "print" statement buffers

Also, here are a few programming language where the default print statement will buffer output when writing to a pipe, and some ways to disable buffering if you want:

- C (disable with `setvbuf`)
- Python (disable with `python -u`, or `PYTHON_UNBUFFERED=1`, or `sys.stdout.reconfigure(line_buffering=False)`, or `print(x, flush=True)`)
- Ruby (disable with `STDOUT.sync = true`)
- Perl (disable with `$| = 1`)

I assume that these languages are designed this way so that the default print function will be fast when you're doing batch processing.

Also whether output is buffered or not might depend on what print function you use, for example in Rust `print!` buffers when writing to a pipe but `println!` will flush its output.

when you press `Ctrl-C` on a pipe, the contents of the buffer are lost

Let's say you're running this command as a hacky way to watch for DNS requests to `example.com`, and you forgot to pass `-l` to tcpdump:

```
sudo tcpdump -ni any port 53 | grep example.com
```

When you press `Ctrl-C`, what happens? In a magical perfect world, what I would *want* to happen is for `tcpdump` to flush its buffer, `grep` would search for `example.com`, and I would see all the output I missed.

But in the real world, what happens is that all the programs get killed and the output in `tcpdump`'s buffer is lost.

I think this problem is probably unavoidable – I spent a little time with `strace` to see how this works and `grep` receives the `SIGINT` before `tcpdump` anyway so even if `tcpdump` tried to flush its buffer `grep` would already be dead.

After a little more investigation, there is a workaround: if you find `tcpdump`'s PID and `kill -TERM $PID`, then tcpdump will flush the buffer so you can see the output. That's kind of a pain but I tested it and it seems to work.

redirecting to a file also buffers

It's not just pipes, this will also buffer:

```
sudo tcpdump -ni any port 53 > output.txt
```

Redirecting to a file doesn't have the same "`Ctrl-C` will totally destroy the contents of the buffer" problem though – in my experience it usually behaves more like you'd want, where the contents of the buffer get written to the file before the program exits. I'm not 100% sure whether this is something you can always rely on or not.

a bunch of potential ways to avoid buffering

Okay, let's talk solutions. Let's say you've run this command or s

```
tail -f /some/log/file | grep thing1 | grep thing2
```

I asked people on Mastodon how they would solve this in practice and there were 5 basic approaches. Here they are:

solution 1: run a program that finishes quickly

Historically my solution to this has been to just avoid the "command writing to pipe slowly" situation completely and instead run a program that will finish quickly like this:

```
cat /some/log/file | grep thing1 | grep thing2 | tail
```

This doesn't do the same thing as the original command but it does mean that you get to avoid thinking about these weird buffering issues.

(you could also do `grep thing1 /some/log/file` but I often prefer to use an "unnecessary" `cat`)

solution 2: remember the "line buffer" flag to grep

You could remember that grep has a flag to avoid buffering and pass it like this:

```
tail -f /some/log/file | grep --line-buffered thing1 | grep thing2
```

solution 3: use awk

Some people said that if they're specifically dealing with a multiple greps situation, they'll rewrite it to use a single `awk` instead, like this:

```
tail -f /some/log/file |  awk '/thing1/ && /thing2/'
```

Or you would write a more complicated `grep`, like this:

```
tail -f /some/log/file |  grep -E 'thing1.*thing2'
```

(`awk` also buffers, so for this to work you'll want `awk` to be the last command in the pipeline)

solution 4: use `stdbuf`

`stdbuf` uses LD_PRELOAD to turn off libc's buffering, and you can use it to turn off output buffering like this:

```
tail -f /some/log/file | stdbuf -o0 grep thing1 | grep thing2
```

Like any `LD_PRELOAD` solution it's a bit unreliable – it doesn't work on static binaries, I think won't work if the program isn't using libc's buffering, and doesn't always work on Mac OS. Harry Marr has a really nice How stdbuf works post.

solution 5: use `unbuffer`

`unbuffer program` will force the program's output to be a TTY, which means that it'll behave the way it normally would on a TTY (less buffering, colour output, etc). You could use it in this example like this:

```
tail -f /some/log/file | unbuffer grep thing1 | grep thing2
```

Unlike `stdbuf` it will always work, though it might have unwanted side effects, for example `grep thing1`'s will also colour matches.

If you want to install unbuffer, it's in the `expect` package.

that's all the solutions I know about!

It's a bit hard for me to say which one is "best", I think personally I'm mostly likely to use `unbuffer` because I know it's always going to work.

If I learn about more solutions I'll try to add them to this post.

I'm not really sure how often this comes up

I think it's not very common for me to have a program that slowly trickles data into a pipe like this, normally if I'm using a pipe a bunch of data gets written very quickly, processed by everything in the pipeline, and then everything exits. The only examples I can come up with right now are:

- tcpdump
- `tail -f`
- watching log files in a different way like with `kubectl logs`
- the output of a slow computation

what if there were an environment variable to disable buffering?

I think it would be cool if there were a standard environment variable to turn off buffering, like `PYTHON_UNBUFFERED` in Python. I got this idea from a couple of blog posts by Mark Dominus in 2018. Maybe `NO_BUFFER` like NO_COLOR?

The design seems tricky to get right; Mark points out that NETBSD has environment variables called `STDBUF`, `STDBUF1`, etc which gives you a ton of control over buffering but I imagine most developers don't want to implement many different environment variables to handle a relatively minor edge case.

I'm also curious about whether there are any programs that just automatically flush their output buffers after some period of time (like 1 second). It feels like it would be nice in theory but I can't think of any program that does that so I imagine there are some downsides.

stuff I left out

Some things I didn't talk about in this post since these posts have been getting pretty long recently and seriously does anyone REALLY want to read 3000 words about buffering?

- the difference between line buffering and having totally unbuffered output
- how buffering to stderr is different from buffering to stdout
- this post is only about buffering that happens **inside the program**, your operating system's TTY driver also does a little bit of buffering sometimes
- other reasons you might need to flush your output other than "you're writing to a pipe"