

Vte line-rewrapping ring design

Behdad Esfahbod

Introduction

Context for this document is in [GNOME bug 336238](#).

The Vte ring is the data-structure holding the vte buffer scrollbar data. It's a "ring" in that appending new items will pop the old ones out. The current design is around the idea of abstraction of *lines*, physical lines in the buffer.

For compactness, we store text as UTF-8, and cell attributes as a run-length-encoded stream. As such, lines contain a variable number of bytes. As a result, designing a ring is not as simple as allocating the ring's maximal memory and looping around it as a classic ring will do. Since you cannot just "discard the beginning bytes" on a memory region or a file, we use a two-magazine design: we fill in the first magazine until it's full of enough data for as much scrollbar as we need, we then switch to the second magazine, discarding whatever was in it. When we fill the second magazine of enough data for a full scrollbar, we discard the first magazine and start filling it again. At any time, we have more than one full scrollbar's worth of data. This design means that we may store up to twice of what we actually need at any time, but that's the price we pay for simplicity.

The current design is around byte-streams. The ring itself uses three bytestreams: one for Unicode text, stored as plain UTF-8; cell attributes, storing only changes in cell attributes (kinda similar to run-length encoding); and line descriptors, which basically store the start offset of the line in the text and attributes streams. I designed this with line-rewrapping in mind, such that rewrapping the buffer would leave the text and attributes streams intact and would only need reencode the line descriptors. The aim of this document is to try to come up with a line descriptor design that wouldn't need reencoding to being with.

Finally, the bytestreams are currently implemented as data structures backed by two POSIX files for the two magazines. The files are created as temp files and immediately unlinked after opening. The design goal was to add a few adaptors to be used before data actually hits the file bytestreams. In particular: a zlib compress()/decompress() adaptor to further reduce our footprint (though, being file-backed, this is less urgent), and, for [security reasons](#), an encryption adaptor

that would encrypt/decrypt data using industry-strength algorithms (AES256?) and a per-terminal random key. The encryption adaptor is more urgent and I encourage people to go ahead and write it. Finally, either of these may serve as a batching adaptor, so we may not need a separate one, but otherwise, we also may want a batching buffer adaptor such that we don't cause disk writes with each line going into the buffer. That said, I wish the kernel had a way for us to tell it "it's alright to keep data for this file in the buffer indefinitely; don't rush spinning the disk plates." Finally, we may also want to have a memory-backed bytestream implementation that `mlock()`'s memory regions, to make sure buffer data never leaves main memory. This may be used to mark security-sensitive buffers, though how to do that short of adding an app setting is not clear to me.

With this introduction behind us, let's talk about rewrapping buffer design.

Design

What we need from the buffer, other than appending data to it, is to request for a line at a given index, given a certain terminal width, and asking for the total number of lines. The ring size is also currently specified as a number of lines, though it doesn't have to be like that. We may very well limit ring size in megabytes or whatever else we desire.

Note that lines can be *hard-wrapped* if a newline preceded them, or *soft-wrapped* if there was no newline. When terminal width changes, it's the soft-wrapped lines that move around, merging with previous lines and then cut again at the new terminal width. As such, let's define a paragraph as the maximal set of lines where first one is hard-wrapped and the following ones are soft-wrapped. The trick to have a buffer that doesn't need explicit rewrapping is to keep paragraphs, not lines, in the buffer. We then need an efficient way to translate between paragraphs and lines given a certain terminal width. Let's imagine that we have an efficient way to answer "how many lines" for a paragraph given a certain width, and that we can also efficiently fetch a certain line in the paragraph. Note that this is not hard to do for small paragraphs, but if you have a looooong paragraph (eg. a 10mb paragraph) then this in itself becomes a problem, short of adding fancy datastructures to the paragraph level. But for now we assume that paragraphs are short and we can brute-force things.

Let me add some discussion for the intra-paragraph stuff, even if we imagined it's trivial. There are two things that make the paragraph to line mapping tricky: tab characters, and CJK double-width characters. Without those, some of the paragraph-to-line operations would be linear and trivial: number of lines in the paragraph? Easy: divide paragraph length by the terminal width and round up, etc. But in presence of tabs and double-width characters we have to scan things to map. What's more interesting for the next-level design is the simple "how many lines per paragraph" question answered over arbitrary widths. This function, let's call it *n-lines*, is an integer non-

increasing function over the width. For now, let's assume that we can efficiently represent this function, for some definition of efficient.

Given the n-lines function for the paragraph, we can build a ring line descriptor system like this: the ring itself will consist of a list of paragraphs. Each paragraph has an n-lines structure. We impose a B-tree structure over the paragraphs, where each node in the B-tree also has the n-lines function of all the paragraphs in it combined. In this manner, the question of "give me line X for width W" can be efficiently implemented in $\log(m)$ calls to n-lines functions where m is the number of paragraphs. If designed with the right size parameters, it may be fine to keep the B-tree in-memory, as opposed to serializing it to a bytestream. This will significantly simplify the implementation.

The way the ring is accessed is very predictable: access has extreme locality properties: most accesses are for rendering the terminal, and those access lines laid after each other at one certain area in the buffer. As such, a cursor data-structure may speed up access, though that must be considered only if profiling shows that speeding up is necessary.

With the high-level design out of the way, let's think about the n-lines implementation.

N-lines

We want this to be efficient in a few ways:

- Memory efficient; consuming a few handfuls of bytes at most, for most cases,
- Fast to call,
- Ideally, for ASCII-only paragraphs, consume zero bytes and be super-fast,

The last item is easy to achieve: if paragraph has no peculiarities (no tabs, no double-width chars), the mapping is trivial and linear. Non-linearity is introduced if either tabs or double-width chars exist, OR, if this is a sum of multiple n-lines (ie. includes multiple paragraphs). This assumes that we know the number of cells / characters in the range. But we can keep that as part of the n-lines datastructure.

Call the previous thing the "linear" n-lines. To store a general n-lines function we can encode its difference from the linear n-lines. ie. you compute the linear output, then add a number to it. The number is an increasing function of width, and we can encode that as a list of (width,delta) items. You'd then bsearch through the list for your width, find the delta, and add it to the linear result.

TODO further discussion re above design. Though, at this point, perhaps best way to assess it is to prototype it.

Further Thoughts

Thinking about this design, I like to note that it can be extended to not care about paragraphs per se, but just about the stream in general. I.e. the problem with long-paragraphs can be obviated by keeping the total stream in a B-tree, with n-lines functions. Sure, lines now may span multiple B-tree nodes, but that doesn't sound like a huge problem, as long as the n-lines function is still efficient to implement.