```
library(Rcpp)

vapply(
  X = fils,
  FUN = cpp_read_file, # see previous post for the source for this C++ Rcpp
function
  FUN.VALUE = character(1),
  USE.NAMES = FALSE
) -> hdrs

head(hdrs, 2)
## [1] "HTTP/1.1 200 OK\r\nDate: Mon, 08 Jun 2020 14:40:45 GMT\r\nServer:
Apache\r\nLast-Modified: Sun, 26 Apr 2020 00:06:47 GMT\r\nETag: \"ace-
ec1a0-5a4265fd413c0\"\r\nAccept-Ranges: bytes\r\nContent-Length: 967072\r\nX-
Frame-Options: SAMEORIGIN\r\nContent-Type: application/x-msdownload\r\n\r\n"
## [2] "HTTP/1.1 200 OK\r\nDate: Mon, 08 Jun 2020 14:43:46 GMT\r\nServer:
Apache\r\nLast-Modified: Wed, 05 Jun 2019 03:52:22 GMT\r\nETag: \"423-
d99a0-58a8b864f8980\"\r\nAccept-Ranges: bytes\r\nContent-Length: 891296\r\nX-
XSS-Protection: 1; mode=block\r\nX-Frame-Options: SAMEORIGIN\r\nContent-Type:
application/x-msdownload\r\n\r\n"
```

However, I need the headers and values broken out so I can eventually get to the analysis I need to do, and a data frame of name/value columns would be the most helpful format. We'll use {stringi} to help us build a function (explanation of what it's doing is in comment annotations) that turns each unkempt string into a very kempt data frame:

```
library(stringi)

parse_headers <- function(x) {

  # split lines from into a character vector
  split_hdrs <- stri_split_lines(x, omit_empty = TRUE)

  lapply(split_hdrs, function(lines) {

    # we don't care about the HTTP x/x ...
    lines <- lines[-1]

    # make a matrix out of found NAME: VALUE
    hdrs <- stri_match_first_regex(lines, "^([^:]*):\\s*(.*)$")

    if (nrow(hdrs) > 0) { # if we have any
      data.frame(
        name = stri_replace_all_fixed(stri_trans_tolower(hdrs[,2]), "-", "_"),
        value = hdrs[,3]
      )
    } else { # if we don't have any
      NULL
    }

  })

}

parse_headers(hdrs[1:3])
## [[1]]
##                name                         value
## 1              date Mon, 08 Jun 2020 14:40:45 GMT
## 2            server                        Apache
## 3     last_modified Sun, 26 Apr 2020 00:06:47 GMT
## 4              etag     "ace-ec1a0-5a4265fd413c0"
## 5     accept_ranges                         bytes
## 6    content_length                        967072
## 7   x_frame_options                    SAMEORIGIN
## 8      content_type      application/x-msdownload
##
## [[2]]
##                name                         value
## 1              date Mon, 08 Jun 2020 14:43:46 GMT
## 2            server                        Apache
## 3     last_modified Wed, 05 Jun 2019 03:52:22 GMT
## 4              etag     "423-d99a0-58a8b864f8980"
## 5     accept_ranges                         bytes
## 6    content_length                        891296
## 7  x_xss_protection                 1; mode=block
## 8   x_frame_options                    SAMEORIGIN
## 9      content_type      application/x-msdownload
##
## [[3]]
##           name                         value
## 1         date Mon, 08 Jun 2020 14:23:53 GMT
## 2       server                        Apache
## 3 content_type text/html; charset=iso-8859-1

parse_header(hdrs[1])
##                name                         value
## 1              date Mon, 08 Jun 2020 14:40:45 GMT
## 2            server                        Apache
## 3     last_modified Sun, 26 Apr 2020 00:06:47 GMT
## 4              etag     "ace-ec1a0-5a4265fd413c0"
## 5     accept_ranges                         bytes
## 6    content_length                        967072
## 7   x_frame_options                    SAMEORIGIN
## 8      content_type      application/x-msdownload
```

Unfortunately, this takes almost 16 painful seconds to crunch through the ~75K text entries:

```
system.time(tmp <- parse_headers(hdrs))
##   user  system elapsed
## 15.033   0.097  15.227
```

as each call can be near 150 microseconds:

```
microbenchmark(
  ph = parse_headers(hdrs[1]),
  times = 1000,
  control = list(warmup = 100)
)
## Unit: microseconds
##  expr     min       lq     mean  median      uq     max neval
##    ph 143.328 146.8995 154.8609 148.361 158.121 415.332  1000
```

A big reason it takes so long is the data frame creation. If you've never looked at the source for `data.frame()` have a go at it — https://github.com/wch/r-source/blob/86532f5aa3d9880f4c1c9e74a417005616846a34/src/library/base/R/dataframe.R#L435-L603 — before continuing.

Back? Great! The {base} `data.frame()` has tons of guard rails to make sure you're getting what you think you asked for across a myriad of use cases. I learned about a trick to make data frame creation faster when I started playing with {ggplot2} source. Said trick has virtually no guard rails — it just adds a class, and `row.names` attribute to a `list` — so you really should only use it in cases like this where you have a very good idea of the structure and values of the data frame you're making. Here's an even more simplified version of the function in the {ggplot2} source:

```
fast_frame <- function(x = list()) {

  lengths <- vapply(x, length, integer(1))
  n <- if (length(x) == 0 || min(lengths) == 0) 0 else max(lengths)
  class(x) <- "data.frame"
  attr(x, "row.names") <- .set_row_names(n) # help(.set_row_names) for info

  x

}
```

Now, we'll change `parse_headers()` a bit to use that function instead of `data.frame()`:

```
parse_headers <- function(x) {

  # split lines from into a character vector
  split_hdrs <- stri_split_lines(x, omit_empty = TRUE)

  lapply(split_hdrs, function(lines) {

    # we don't care about the HTTP x/x ...
    lines <- lines[-1]

    # make a matrix out of found NAME: VALUE
    hdrs <- stri_match_first_regex(lines, "^([^:]*):\\s*(.*)$")

    if (nrow(hdrs) > 0) { # if we have any
      fast_frame(
        list(
          name = stri_replace_all_fixed(stri_trans_tolower(hdrs[,2]), "-", "_"),
          value = hdrs[,3]
        )
      )
    } else { # if we don't have any
      NULL
    }

  })

}
```

Note that we had to pass in a `list()` to it vs bare name/value vectors.

How much faster is it? Quite a bit:

```
microbenchmark(
  ph = parse_headers(hdrs[1]),
  times = 1000,
  control = list(warmup = 100)
)
## Unit: microseconds
##  expr   min      lq     mean median       uq      max neval
##    ph 27.94 28.7205 34.66066 29.024 29.3785 4144.402  1000
```

This speedup means the painful ~15s is now just a tolerable ~3s:

```
system.time(tmp <- parse_headers(hdrs))
##   user  system elapsed
## 2.901   0.011   2.918
```

FIN

Normally, guard rails are awesome, and you can have even more safe code (which means safer and more reproducible analyses) when using {tidyverse} functions. As noted in the previous post, I’m doing a great deal of iterative work, have more than one set of headers I’m crunching on, and am testing out different approaches/theories, so going from 16 seconds to 3 seconds does truly speed up my efforts and has an even bigger impact when I process around 3 million raw header records.