Stupid command-line trick: Counting the number of lines in stdin

devblogs.microsoft.com/oldnewthing/20110825-00

August 25, 2011

Raymond Chen

On unix, you can use `wc -l` to count the number of lines in stdin. Windows doesn't come with `wc`, but there's a sneaky way to count the number of lines anyway:

```
some-command-that-generates-output | find /c /v ""
```

It is a special quirk of the `find` command that the null string is treated as never matching. The `/v` flag reverses the sense of the test, so now it matches everything. And the `/c` flag returns the count.

It's pretty convoluted, but it does work.

(Remember, I provide the occasional tip on batch file programming as a public service to those forced to endure it, not as an endorsement of batch file programming.)

Now come da history: Why does the `find` command say that a null string matches nothing? Mathematically, the null string is a substring of every string, so it should be that if you search for the null string, it matches everything. The reason dates back to the original MS-DOS version of `find.exe`, which according to the comments appears to have been written in 1982. And back then, pretty much all of MS-DOS was written in assembly language. (If you look at your old MS-DOS floppies, you'll find that `find.exe` is under 7KB in size.) Here is the relevant code, though I've done some editing to get rid of distractions like DBCS support.

```
        mov     dx,st_length            ;length of the string arg.
        dec     dx                      ;adjust for later use
        mov     di, line_buffer
lop:
        inc     dx
        mov     si,offset st_buffer     ;pointer to beg. of string argument
comp_next_char:
        lodsb
        cmp     al,byte ptr [di]
        jnz     no_match
        dec     dx
        jz      a_matchk                ; no chars left: a match!
        call    next_char               ; updates di
        jc      no_match                ; end of line reached
        jmp     comp_next_char          ; loop if chars left in arg.
```

If you're rusty on your 8086 assembly language, here's how it goes in pseudocode:

```
 int dx = st_length - 1;
 char *di = line_buffer;
lop:
 dx++;
 char *si = st_buffer;
comp_next_char:
 char al = *si++;
 if (al != *di) goto no_match;
 if (--dx == 0) goto a_matchk;
 if (!next_char(&di)) goto no_match;
 goto comp_next_char;
```

In sort-of-C, the code looks like this:

```
 int l = st_length - 1;
 char *line = line_buffer;
 l++;
 char *string = st_buffer;
 while (*string++ == *line && --l && next_char(&line)) {}
```

The weird `- 1` followed by `l++` is an artifact of code that I deleted, which needed the decremented value. If you prefer, you can look at the code this way:

```
 int l = st_length;
 char *line = line_buffer;
 char *string = st_buffer;
 while (*string++ == *line && --l && next_char(&line)) {}
```

Notice that if the string length is zero, there is an integer underflow, and we end up reading off the end of the buffers. The comparison loop does stop, because we eventually hit bytes that don't match. (No virtual memory here, so there is no page fault when you run off the end of a buffer; you just keep going and reading from other parts of your data segment.)

In other words, due to an integer underflow bug, a string of length zero was treated as if it were a string of length 65536, which doesn't match anywhere in the file.

This bug couldn't be fixed, because by the time you got around to trying, there were already people who discovered this behavior and wrote batch files that relied on it. The bug became a feature.

The integer underflow was fixed, but the code is careful to treat null strings as never matching, in order to preserve existing behavior.

Exercise: Why is the loop label called `lop` instead of `loop`?

Raymond Chen

Follow