m4 Obstack Interleaved Write Bug

John Brzustowski

January 24, 2006

1 Overview

After GNU m4 reaches the end of input and begins to process wrapup text saved by `m4wrap()`, it does not always correctly deal with subsequent calls to `m4wrap()`. Below are details of the misbehaviour, and a proposed solution. A set of patches to m4-1.4.4 is provided separately.

2 Current problematic behaviour

Due to word-size and alignment differences, the tests described below may not expose the buggy behaviour on all platforms. `^D` in what follows represents the system end-of-file character, which prompts m4 to begin processing wrapup text.

Wrapping from within wrapped text can cause an infinite loop:

```
$ m4
m4wrap(‘m4wrap(a)m4wrap(b)’)
^D
=> (infinite loop)
```

instead of the desired (and specified) output `ba`. Here are two examples which might expose the bug on other platforms, especially if large values are passed to `f()`.

A simple countdown function:

```
$ m4
define(‘f’,‘ifelse(
eval(‘$1>0’),
0,
‘m4wrap(0)’,
‘m4wrap($1:)m4wrap(‘f(decr($1))’)’)’)
f(1000)
^D
=> Segmentation fault
```

instead of `1000:999:998:...:2:1:0`

An awkward definition of the factorial function:

```
$ m4
define(‘f’,‘ifelse(
eval(‘$1>1’),
0,
Answer: $2$1=‘eval($2$1) ’,
‘m4wrap(‘f(decr($1),$2$1*)’)’)’)
f(10)
^D
=> NONE:0: m4: INTERNAL ERROR: Input stack botch in peek_input ()
=> Aborted
```

instead of `Answer: 10*9*8*7*6*5*4*3*2*1=3628800`

3 Source of the problem

GNU m4 uses an obstack, input_obstack, for keeping track of text to be processed. This input can come from files, strings, or any text which must be re-read, such as a just-expanded macro. Along the way, any calls to m4wrap() stack their output onto a separate obstack, wrapup_stack. These two stacks have pointers to their top entries in the variables isp and wsp, respectively. When m4 reaches the end of input_obstack, it begins reading wrapup_stack for any wrapup text to be processed, using wrapup_stack exactly as it had previously used input_stack. The pointers isp and wsp can now point to different objects in the *same* obstack.

While processing wrapup_stack, any further calls to m4wrap() also place their output on wrapup_stack, and this can lead to interleaved write access: the input processor might deleted an object from wrapup_stack even if that object is no longer at the top of the stack because a call to m4wrap() has pushed something onto it. This leaves m4wrap()ed entries on wrapup_stack effectively freed (obstack_free() frees a given object and all those above it on the obstack), and subject to overwriting when the input processor next grows an object on wrapup_stack.

Consider this example:

```
$ m4 -dqeat
m4wrap(`format(`%s is good.', m4wrap(`All done!')`My luck')')
^D
=> m4trace: -1- m4wrap(`format(`%s is good.', m4wrap(`All done!')`My luck')')
=>
=> m4trace: -2- m4wrap(`All done!')
=> m4trace: -1- format(`%s is good.', `My luck') -> `My luck is good.'
=> Segmentation fault
```

The interleaved writing occurs as follows:

- m4 reads ^D and repoints its input_stack to the wrapup_stack, which will now also be the input stack. It looks like this:

```
     TOP-OF-STACK
isp->Input Block: format(`%s is good.', m4wrap(`All done!')`My luck')
     BOTTOM-OF-STACK

wsp->NULL
```

- m4 notices the macro call `format()` and begins collecting its arguments
- m4 notices the macro call `m4wrap()` and begins collecting its arguments
- m4 allocates an entry on the input stack for expanding `m4wrap()`, since its expansion will be re-read as input. The stack now looks like:

```
          TOP-OF-STACK
          Input Block: (space for expansion of m4wrap()
     isp->Input Block: format(`%s is good.', m4wrap(`All done!')`My luck')
          BOTTOM-OF-STACK
     wsp->NULL
```

- m4 calls the internal function `m4_m4wrap()` which pushes its argument on the stack, which now looks like:

```
          TOP-OF-STACK
     wsp->Input Block: `All done!'
     isp->Input Block: (space for expansion of m4wrap()
          Input Block: `format(`%s is good.', m4wrap(`All done!')`My luck')'
          BOTTOM-OF-STACK
```

- m4 looks for the next input token, for which it uses `isp`. Since the expansion of `m4wrap()` is empty, the input stack is popped, which deletes everything from `isp` to the top of the stack:

```
          DELETED-STACK-ENTRIES
     wsp->Input Block: `All done!'
          Input Block: (space for expansion of m4wrap()
          TOP-OF-STACK
     isp->Input Block: `format(`%s is good.', m4wrap(`All done!')`My luck')'
          BOTTOM-OF-STACK
```

- m4 finishes collecting the arguments to `format()`, then allocates space on the stack for its expansion:

```
          DELETED-STACK-ENTRIES
     wsp->Input Block: `All done!'
          TOP-OF-STACK
     isp->Input Block: (space for expansion of `format()'
          Input Block: `format(`%s is good.', m4wrap(`All done!')`My luck')'
          BOTTOM-OF-STACK
```

- Now, as `format()` is expanded, the expansion will overwrite the input block pointed to by `wsp`, leading to eventual havoc when the end of input is reached and m4 again seeks to read the wrapped text pointed to by `wsp`.

4 Proposed fix

The GNU documentation is clear on how `m4wrap()` is supposed to behave, so the changes below serve only to make m4 behaviour conform to the specification.

Rather than using statically-allocated obstacks for `input_stack` and `wrapup_stack`, these are dynamically allocated. When m4 is finished processing `input_stack`, it is freed, and the pointer repointed to `wrapup_stack`. A new empty obstack is allocated for `wrapup_stack`, and this is where subsequent `m4wrap()` calls will place their output.

4.1 Changes to functions

- `input.c:`

  Declare `input_stack` and `wrapup_stack` as `obstack *` instead of `obstack`.

  Remove the declaration for `current_input`, which is used an an alias for either `&input_stack` or `&wrapup_stack`. All functions in `input.c` will now directly use `input_stack` instead of `current_input`.

  Inline documentation is changed to reflect this.

- `input.c: input_init()`

  Set `input_stack` and `wrapup_stack` to point to empty, dynamically-allocated obstacks.

- `input.c: pop_wrapup()`

  Free the obstack and its contents pointed to by `input_stack`.

  Assign `wrapup_stack` to `input_stack`.

  Point `wrapup_stack` to a newly allocated and initialized obstack.

5 Discussion

This fix allows the examples agove to perform correctly. I have run the countdown example from above with f(1000000) and monitored the process with `top`. The amount of memory allocated does not increase, despite the repeated allocation and freeing of obstacks. (The factorial example uses an increasing amount of memory, as coded.)

For better or worse, the fix allows a new kind of infinite loop:

```
$ m4
define(`f',`m4wrap(`f')')
f
^D
=> (infinite loop)
```

This is similar to the existing example:

```
$ m4
define(`f',`f')
f
^D
=> (infinite loop)
```

in semantics, but differs in that it repeatedly allocates and frees obstacks for the wrapped text, rather than re-using the same input obstack.