Program Execution Explorer lab

Useful pointers

- Debugging with GDB (2017)
- Arithmetic operations in the GNU Elisp manual (2016)

Background

The name of this assignment comes from the idea that a debugger like GDB is better thought of as a way of exploring program execution histories than merely as a debugger.

Although the GNU Emacs text editor is not intended for high performance numeric computation, its scripting language Elisp is reasonably widely used and Elisp applications need adequate (if not stellar) performance for numeric applications. One such application, GNU Calc, is a desk calculator that does a lot of arithmetic internally. Your goal is to see how much overhead is imposed by Emacs when doing standard arithmetic operations, in particular multiplication with integer arguments, and to think about how to reduce the arithmetic overhead.

Keep a log

Keep a log in the file `pexexlab.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a lab notebook, in which you briefly note down what you did and what happened. It should record not just what worked, but also what *didn't* work.

Gather instruction traces

You can multiply numbers with Emacs from the shell by running a command like this:

```
emacs -Q -batch -eval '(print (* 1250284240 -1844255039))'
```

Gather a trace for the key part of the above test case. This trace should include every instruction in the `Ftimes` function, which is the C implementation of the Elisp `*` function. It should also include every instruction in every function that `Ftimes` calls, either directly or indirectly.

For the purpose of this assignment, a trace is an ASCII text file. Each line corresponds to a single machine instruction executed in turn during the trace. Lines use the following format:

```
5477fd  41 57   push %r15       M8[7fffffffd380]=0 rsp=7fffffffd380
```

Columns should be separated by single tab characters. The first column gives the machine address of the instruction, in hexadecimal (without leading `0x` or excess leading `0`). The second column gives the bytes of the machine instruction, in hexadecimal (without leading `0x`) separated by single spaces. The third column gives the assembly-language equivalent of the second column, using the notation that GDB uses after you execute the command `set disassemble-next-line on`, except with a single space separating each column. The fourth column gives the effect of the instruction on memory and general-purpose registers, again using unsigned hexadecimal in the same style as the first column. The example above stores 0 into the 8-byte word at address 7fffffffd380 (because `r15` happens to be zero); the "8" in "M8" stands for an 8-byte memory access. The example also sets the `rsp` register to 7fffffffd380. List memory modifications in increasing address order, and register modifications in alphabetical order. Traces need not record modifications to status registers such as `rflags`.

To gather information for your trace (which you should put into the file `trace.tr`), use the executable `~eggert/bin64/bin/emacs-25.2` on either `lnxsrv07` or `lnxsrv09`. The corresponding source code can be found in `~eggert/src/emacs-25.2/` (particularly its `src` subdirctory), and the executable was compiled for the x86-64. The above example trace line corresponds to 45 bytes into the machine code for the function `arith_driver`, which in turn corresponds to line 2635 of `~eggert/src/emacs-25.2/src/data.c`.

Examine integer overflow handling

Compile the following function:

```
_Bool
testovf (long n)
{
  return n + 9223372036854775807 < n;
}
```

for the x86-64 in three ways: (1) with `-O2`, (2) with `-O2 -fwrapv`, (3) with `-O2 -fsanitize=undefined`. Compare the resulting assembly-language files, and describe and justify the differences that you see. Put your description into a plain ASCII text file `testovf.txt`.

A few more questions

Answer the following questions, in a plain text file `answers.txt`:

1. Explain why the instructions in the trace did not produce the correct mathematical result. Which instructions caused the problem, exactly?
2. Explain why the shell command `emacs -Q -batch -eval '(print most-negative-fixnum)'` outputs `-2305843009213693952`. Where did the number come from? Explain in terms of the Emacs source code.
3. Explain why the shell command `emacs -Q -batch -eval '(print (* most-positive-fixnum most-positive-fixnum most-positive-fixnum most-positive-fixnum))'` outputs only 1.
4. The Emacs executable was compiled with GCC's `-O2` option. Suppose it had also been compiled with `-fwrapv`. Explain any problems Emacs would run into, or if there would not be a problem explain why not.
5. There is a recently-discovered security vulnerability in Emacs 25.2, which you can exercise by running `emacs -Q -batch -eval '(print (format-time-string "%Y-%m-%d %H:%M:%S %Z" nil (concat (make-string 1000 ?X) "0")))'`. Briefly describe the bug's low-level manifestation by crashing Emacs, using GDB's `backtrace` command, and following up with any other GDB commands that you think might be of interest.

Submit

Submit a compressed tarball `pexex.tgz` containing the files mentioned above, namely `pexexlab.txt`, `trace.tr`, `testovf.txt`, and `answers.txt`.

$Id: pexexlab.html,v 1.9 2017/04/28 04:49:44 eggert Exp $