Debugging Lab

A `gdb` crash course

Introduction

Debuggers are invaluable tools for every serious programmer. Therefore, it is important that you learn to master a debugger. This assignment is an introduction to using the *gdb* debugger on Linux systems. Many other debuggers exist, such as *lldb* which is now commonly used on Apple systems and for example the Visual Studio debugger. Although other debuggers work with different commands, or even have a fully graphical user interface, the basic idea remains the same. A debugger allows you to "step" through the program and inspect the current state of the process at any point in time. Most debuggers also allow you to set "breakpoints", which interrupt the program at a certain function (eventually when a certain condition holds) such that the process' state can be inspected. There even exist debuggers which can rewind (step backwards)!

gdb is a command-line debugger. This means you are presented with an input prompt at which point you should enter a command. *gdb* is typically started with the program to debug as argument:

```
gdb ./myprogram
```

Once *gdb* has started, you can perform some preparations, such as setting breakpoints (see later on). When you are ready to start execution of your program, you enter the command `run`.

At any time you can interrupt the execution of your program by pressing *Ctrl+C*. *gdb* will take over control and show the prompt. You can now inspect the current state of your program (`print` command) and manipulate breakpoints. To continue program execution, enter the `continue` command.

Instead of continuing the program until the next breakpoint or manual interruption, you can also step through a program. The `next` command executes instructions until the next source code line is reached. So, the command executes one line of source code. If the line contains a function call, this entire function call is executed. Sometimes (actually, quite often), you want to step *into* such functions and step through the function line by line. In that case, use the `step` command instead of `next`.

Inspecting Program State

With *gdb* you can inspect the current state of a program in execution. With the `print` command you can inspect values of variables and evaluate expressions. You may also use the shorthand `p`. By appending `/` and a formatting specifier, you can change the way *gdb* outputs the value. For example, `/x` results in hexadecimal output, `/t` in binary output. A complete overview of the possible "format letters" can be found with the `help x` command. Some examples of the use of `print`:

```
(gdb) p a
$1 = 13
(gdb) p/x a
$2 = 0xd
(gdb) p a + 14
$3 = 27
(gdb) p/t a + 14
$4 = 11011
(gdb) p $4 * 7
$5 = 189
(gdb) p str
$6 = 0x40061c "hello world"
(gdb) p str[10]
$7 = 100 'd'
(gdb) p *str@6
$8 = "hello "
(gdb) p/x *str@6
$9 = {0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x20}
```

You can also directly inspect the memory belonging to the process. This is done using the `x` command ("examine"). Similar to `print` the format letters can be used to specify how the memory contents should be formatted. With the number after the slash you specify how many elements should be printed. Example using the address of `str` from the above example as memory address:

```
(gdb) x/16x 0x40061c
0x40061c: 0x6c6c6568 0x6f77206f 0x00646c72 0x25206425
0x40062c: 0x00000a73 0x3b031b01 0x0000002c 0x00000004
0x40063c: 0xfffffdb0 0x00000048 0xfffffec4 0x00000070
0x40064c: 0xffffff00 0x00000090 0xffffff90 0x000000b8
(gdb) x/16b 0x40061c
0x40061c: 0x68 0x65 0x6c 0x6c 0x6f 0x20 0x77 0x6f
0x400624: 0x72 0x6c 0x64 0x00 0x25 0x64 0x20 0x25
(gdb) x/16c 0x40061c
0x40061c: 104 'h'101 'e'108 'l'108 'l'111 'o'32 ' '119 'w'111 'o'
0x400624: 114 'r'108 'l'100 'd'0 '\000'37 '%'100 'd'32 ' '37 '%'
```

Finally, another important property of program state is where exactly we are in the execution of the program (program counter). And how did we arrive at this location? Which function calls were done and where? This can be seen with the `backtrace` command. It outputs a list of stack frames. These frames are all numbered. Using the `frame` command followed by a (valid) frame number, you can "select" this frame. The context is then switched to this frame and you can inspect values of variables that exist within that stack frame.

Dealing with breakpoints

Working with breakpoints is an essential part of debugging. With breakpoints, you can specify locations in the program at which point execution must be interrupted. When the execution is interrupted you can inspect the program state using the *gdb* prompt. Breakpoints are set using

the `break` command. As argument you must specify a location, this can be the name of a function defined your program, or a source code location for example `file.c:354` for line 354 in `file.c`. Note that for *gdb* to be able to find these locations, it is essential that your program has been compiled with debugging information using the `-g` switch to the compiler.

Once added, a breakpoint is assigned a number. This number is used to enable and disable breakpoints with the commands `enable` and `disable`. `info break` lists the currently defined breakpoints. Using `delete` followed by a number a breakpoint is deleted.

Breakpoints can be made conditional: in this case, the execution is only interrupted if the condition evaluates to true. A condition can be set using the `condition` command, for example:

```
cond 3 a > 103
```

sets the condition `a > 103` for breakpoint 3.

Finally, `list` is a very useful tool when looking for a location where to set a breakpoint. With `list` you can let *gdb* show source code. When used without argument, *gdb* shows source code around the current location of the program counter. As argument you may enter a function name, or location such as `file.c:345`, similar to the location that must be specified when setting a breakpoint.

Command overview

The following list summarizes a number of often used *gdb* commands. The shorthand for each command is listed with parentheses.

- `print` (`p`) – show the result of the specified expression (for instance the name of a variable).
- `x` – show contents at the specified memory address.
- `run` (`r`) – run the loaded program. Optionally, arguments to the program to run may be specified.
- `continue` (`c`) – continue running the program.
- `next` (`n`) – execute until next line of source code, execute function calls without stopping.
- `step` (`s`) – execute until next line of source code, also steps into function calls.
- `finish` – execute until the end of the current function.
- `until` – if the current line is a loop, `until` will continue execution until the end of the loop. So, you jump over the loop.
- `list` (`l`) – list source code.
- `backtrace` (`bt`) – show a trace of all stack traces (a listing of all current function calls).
- `frame` (`f`) – show the currently selected stack frame, or select another stack frame by number.
- `break` (`b`) – set a breakpoint.
- `enable` (`ena`) – enable a breakpoint by number.
- `disable` (`dis`) – disable a breakpoint by number.
- `delete` (`del`) – delete a breakpoint by number.
- `info break` – list currently defined breakpoints.
- `help` (`h`) – help!

Memory Debugging

Bugs that are triggered by corruption of program memory are incredibly hard to track down, even when using *gdb*. This is caused by the fact that memory corruption modifies program state inadvertently and unexpectedly. Fortunately, tools exist to aid with "memory debugging". One of the most widely tools used is *valgrind*. Your program does not have to be modified in order to use *valgrind*. Instead, *valgrind* interprets and executes the instructions of your program, transparently adding instrumentation to memory calls such as *malloc*. As a consequence, your program will run noticeably slower. This is usually not a problem, but could be if the bug to the solved is timing-dependent.

You can invoke *valgrind* simply by adding the command `valgrind` in front of the command-line to launch your program, for example:

```
valgrind ./myprogram arg1 arg2
```

valgrind outputs detailed reports about any problem it encounters. If the program includes debugging information, a symbolic stack trace is shown of the location where the problem was detected. We leave the interpretation of *valgrind*'s output as a exercise for the reader.

Exercises

To get some practice using *gdb* we have produced three buggy programs that can be obtained from the course website. Compile the programs using `make`. Note that all programs compile without any compiler errors or warnings. Try to run the programs and fix all bugs. The expected behavior of the programs is briefly described below.

Note that there is nothing you have to hand in and there is no deadline. This is just a practice session to help you get accustomed to the use of *gdb*.

Exercise 1. The first program outputs a listing of the specified directory. The directory name is prepended to each file that is listed. By default `/usr/bin` is listed, but another directory may be specified as command-line argument. Try to run for `/`, `/usr`, `~` and finally `/usr/bin`.

Exercise 2. Joe thought it was a good idea to practice his pointer arithmetic skills. As an example program he took dense matrix multiplication. Can you modify the program and in particular get the pointer arithmetic right such that the program works correctly (don't just replace the arithmetic with normal array subscription, practice your pointer skills!)? When the program finishes without generating any output, the program works correctly.

Exercise 3. The program expects a list of numbers (one number per line) as input. It then creates a binary tree and performs an in-order traversal. In this case, the numbers are visited in sorted order. During the traversal, the numbers are written to an output buffer. Unfortunately, the implementation has some problems. You can input some numbers yourself to test the program, we have also provided an example input which can be used as follows: `./bug3 < bug3input.txt`.