Homework 8 - Build your own debugger

In this homework we'll be building a stripped down version of gdb. The template solution comes with a library for reading debugging symbols (dwarf_symbol_reader) and for processing user input (debugger_interface), allowing you to focus on developing the features.

Initial Setup

You'll need a few libraries first. Run the following command before trying to compile the template...

```
sudo apt-get install libelf-dev libdwarf-dev dwarfdump
```

dwarfdump is a program that displays the dwarf debugging symbols for a program. It may be helpful to you for this assignment.

API INFO

You'll have to use the ptrace API for this assignment along with a couple of libraries that I wrote. Familiarize yourself with these things **BEFORE** starting the assignment! I suggest you peek at the .c and .h files also, just to get a flavor of what is going on.

Debugger Interface (debugger_interface.h)

This module handles the user input to the debugger. Each command from the user has a command type and a value associated with it. For instance, if the user types "b 7" then the command type is CMD_TYPE_BREAKPOINT and the value is the line number where we want to set the breakpoint.

Values can be strings (for instance the name of a variable for printing) or integers so we use a union type to represent it. The union type in the user_command struct is called value. Below is an example of how to access the line_number from a user_command.

```
user_command * cmd = debugger_interface_get_cmd();
//now handle the input
switch(cmd->command_type){
    case CMD_TYPE_BREAKPOINT:
      set_breakpoint_from_line_number(cmd->value.line_number);
      break;
    case CMD_TYPE_PRINT:
      printf("printing not implemented yet\n");
      break;
    case CMD_TYPE_CONTINUE:
      //tell the child to keep going
      break;
  }
```

dwarf_symbol_reader.h

This module allows you to get info about the program that you are debugging (assuming that program was compiled with dwarf debugging symbols). This includes such info as function names, function starting and ending addresses, variable names and locations, etc... The dwarf_symbol_reader.h file has lots of comments so make sure to take a look in there. **For all of the parts of the assignment you will need to utilize this library!**.

Dwarf considers each object file a "compilation unit". For this assignment we'll only have a single compilation unit because the programs we will be using are very simple. Many of the functions in dwarf_symbol_reader.h take a compilation unit, so you'll want to grab the compilation unit at the beginning of the debugging process using the function **dwarf_get_compilation_unit**.

A **die** (debugging information entries) contains the debugging information for a given entity in a program. So a function would have a die, a variable would have a die, and so forth. In fact, the whole compilation unit also has a die and is available in the **struct dwarf_compilation_unit** type as **root_die**. The dies are organized into a tree with the compilation unit as a root. So, if variable x is in function fun1, then x's die is a child of fun1 . Most of the functions in dwarf_symbol_reader.h take a die as a parameter.

One of the main things that you can do with these functions is iterate through a bunch of dies. So lets say you wanted to loop through all the function dies in your program. You would do the following

```
//start with the first child of the root die
Dwarf_Die func_die =  dwarf_get_next_function(compilation_unit->root_die, compilation_unit);

```

```
while(func_die){
    //do something with the function die
    //...
    //now get the next one
    func_die =  dwarf_get_next_function(func_die, compilation_unit);
}
```

There are similar iterator functions available for variables.

Using Ptrace

The Ptrace library allows a debugging program to have total control over the debuggee. Using Ptrace, our debugger (via the operating system) can modify a program's instructions, read a program's memory, stop it, start it, single step it, etc... The man page for Ptrace is great and is linked below in the references. **You will need to make at least 1 Ptrace call for every part of this homework**.

a. Setting a breakpoint (2 pts)

The first thing you will need to do is set a breakpoint and have the program stop at that line. To do this you will have to modify the right instruction by replacing the low order byte with an int3 (0xCC) op code. Make sure to store away the old instruction so that you can put it back once the breakpoint has been hit. The users interaction with this functionality should look like this...

```
385db>> b 7
breakpoint 1 set at line 7
385db>> c
continuing...
breakpoint 1 hit
385db>>
```

b. Where am I? (2 pts)

When debugging it is always helpful to know where you are in a certain file. For part 1 you will print out the line you are currently on and the file name. (HINT: use the instruction pointer from ptrace to figure out the line). Here is the intended output...

```
385db>> where
line 1 in prog1.c
```

c. Persistent breakpoints (2 pts)

When you hit a breakpoint you must take the original instruction and put it back, thus removing your breakpoint. For part b we now make our breakpoints persistent so that they stick around. (Hint: take a look at single stepping with ptrace). The user should be able to hit the same breakpoint multiple times without having to set it more than once.

d. Multiple breakpoints (1 pt)

Its not very fun for the user to only be able to set one breakpoint at a time. Now allow the user to have multiple breakpoints set concurrently. Here's an example of the expected output.

```
385db>> b 7
breakpoint 1 set at line 7
385db>> b 8
breakpoint 2 set at line 8
385db>>c
hit breakpoint 1
385db>>c
hit breakpoint 2
```

e. Printing Variables (3 pts)

Being able to inspect the value of a variable is an important part of debugging. For part e we will now implement the printing of variables. The p command will take two arguments: the first is the format specifier (/d for integer in decimal, /c for character, etc...) and the second is the variable name. **I will only test that your program can print integers and characters**. Here's the expected output.

```
385db>> b 7
breakpoint 1 set at line 7
385db>> p /d x
101
385db>> c
```

f. Printing out strings (3 bonus pts)

If you can print out a single value it should be too hard to also print out a string of characters. For three points bonus, modify your program to do just that.

```
385db>> p /s trueStr
"CS 385 is so much fun!"
385db>> c
```

?

g. Generic printing of variables (5 bonus pts)

If you run dwarfdump , you will see the dies that represent each variable. You will also notice that each die has a type field that references a type die which is also displayed by dwarfdump. You can use this information to figure out a variable's type and then print out that variable without the format specifier.

To get the bonus points you must utilize libdwarf.h to read the type information from the variable die. This functionality is **not** available in the dwarf_symbol_reader.h file. You must implement it yourself! Look at the code in dwarf_symbol_reader.c for examples of how to use the libdwarf library. If you decide to tackle this, make sure to let me know before I grade your solution!

h. (WEEK 3) Printing the line of code (3 pts)

Previously the "where" command would just print the line number and the file name. Now, change "where" to also print out the line of code. For this you don't need to use dwarf symbols. Just use basic C file reading to find the line.

```
385db>> where
line 4 in prog1.c

for (; i < 10; ++i){
```

?

i. (WEEK 3) Printing out the stack trace (1 pt)

This one should be fun. Now allow the users to see a very simple stack trace after typing "bt". This should work for any number of function calls, including recursive ones. Here's the output we should see given the following debuggee program...

```
void h(int y){
     printf("hello there: %d", y);
}

void g(int x){
      h(x);
}

int main(){
     g(10);
}
```

?

```
385db>> b 2
breakpoint 0 set at line 2
385db>> c
hit breakpoint 0
385db>> bt
#0 h()
#1 g()
#2 main()
```

?

j. (WEEK 3) A better stack trace (1 pt)

And last but not least, make your stack trace look more like gdb's by adding the source file, line number and the input parameters. If you did not do the generic printing, then you can just assume that all input parameters are integers. So for the above example, the stack trace should look like this

```
385db>> b 2
breakpoint 0 set at line 2
385db>> c
hit breakpoint 0
385db>> bt
#0 h(y=10) at tests/prog5.c:2
#1 g(x=10) at tests/prog5.c:6
#2 main() at tests/prog5.c:10
```

?

References

1.) Take a look at this blog entry...it should take you a long way How a Debugger Works

2.) The Ptrace man page

3.) An in depth look into dwarf Debugging formats DWARF and STAB

-- TimothyMerrifield - 2012-03-12

Copyright 2012 The Board of Trustees
of the University of Illinois.webmaster@cs.uic.edu

WISEST
Helping Women Faculty Advance
Funded by NSF

MAKE A GIFT TO THE DEPARTMENT OF COMPUTER SCIENCE

Open House Information