Source level debugging

October 18, 2016

Source level debugging

- Source debugging is a nice tool for debugging execution problems; it can be particularly useful when working with crashed programs that leave a dump file.
- To enable source debugging with the gcc/clang families of compilers, you can use the -g option.

Source level debugging

- Also, most assemblers support the underlying DWARF format; generally you have to add something like -g dwarf2 to be specific. (Also see Wikipedia DWARF entry)

Source level debugging

- The symbol table information includes the correspondence between
  - statements in the source and location of that code in the executable
  - variables in the source and locations of those variables in memory

GDB: the Gnu debugger

- GDB is a line-oriented debugger where actions are initiated by typing in commands at a prompt.
- It can be invoked for executables created by gcc, clang, and nasm/yasm.

GDB: the Gnu debugger

- General capabilities
  - Starting and exiting your program under the debugger
  - Pausing and continuing execution while in the debugger
  - Examining the state of your program
  - Changing the state of your program
  - Viewing registers and memory

Starting and stopping GDB

- You can start gdb along these lines:

```
gdb YOURPROGRAM [core|pid]
```

- If you don't specify a core file or a process id, then you can start a new process executing your YOURPROGRAM with the run command.

Starting and stopping GDB

- You can give the run command options in the same manner that you would at a bash prompt:

```
run 123 > /tmp/out
```

- You can exit gdb with the quit command.

Stopping and continuation

- You can set and remove breakpoints
- You can also step through an execution, or just continue it.

Setting and removing breakpoints

- You can set a breakpoint to stop either when a certain location in the source is reached, or when a condition is met.
- The general form is

```
break [SOMEFUNCTION|SOMELINENUM] [if SOMECONDITION]
```

Setting and removing breakpoints

- Specifying just break will set a breakpoint at your current location.
- You can remove a breakpoint with

```
delete BREAKPOINT
```

Examples

```
(gdb) break          Sets a brkpt at the current line
(gdb) break 50       Sets a brkpt at line 50 of the current f
(gdb) break main     Sets a brkpt at routine main()
(gdb) break 10 if i == 66
                     Break execution at line 10 if the variabl
                     i has the value 10
(gdb) delete 3       Delete the third breakpoint
(gdb) delete         Delete all breakpoints
(gdb) info breakpoint
                     List all breakpoints
```

Stepping into a function

You can step into a function with “s”, or just go the next line of code with “n”

The general form is

```
step [N]
```

where N indicates the number of steps to take, defaulting to 1 if not specified. Execution will not continue through a breakpoint — or program termination. ;-)

Nexting through execution

Of course, often you don't want to step *into* a function. You can use the next command to go to the next statement rather than stepping into a function specified on the current line.

`next [N]`

Finishing a function

It's pretty easy to accidentally step into library code that you don't have the source for; "finish" will get you out of that problem:

```
$ gdb hello_world
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.2) 7.7.1
Reading symbols from hello_world...done.
(gdb) break main
Breakpoint 1 at 0x4004f0: file hello_world.c, line 5.
(gdb) run
Starting program: hello_world
```

Finishing a function

```
Breakpoint 1, main () at hello_world.c:5
5     printf("Hello world\n");
(gdb) s
__printf (format=0x4005a4 "Hello world\n") at printf.c:28
28  printf.c: No such file or directory.
(gdb) s
32  in printf.c
```

Finishing a function

```
(gdb) finish
Run till exit from #0  __printf (format=0x4005a4
                                 "Hello world\n")
    at printf.c:32
Hello world
0x00000000004004f7 in main () at hello_world.c:5
5     printf("Hello world\n");
Value returned is $1 = 12
(gdb)
```

Until the end of a loop

You can use the until command to execute your program until it reaches a source line greater than the one that you are currently on. If you are not at a "jump back", then is the same as the next command. If you are at a "jump back" point such as in the last statement of a while loop, then this will let you execute until the point that you have exited the loop.

Examining the state of your program

- Listing source code
- Printing the values of expressions
- Displaying the values of expressions
- Printing a stack trace
- Switching context in a trace
- Showing the contents of memory
- Showing the contents of registers

Listing source code

You can list source code at a specified line or function

The general forms are

```
list [[FILENAME:]LINENUM[,LINENUM]]
list [[FILENAME:]FUNCTIONNAME]
```

Listing source code

If you don't specify anything, then you will get 10 lines from the current program location, or 10 more lines if you have already listed the current program location.

```
(gdb) list      List 10 lines from
                the current location
(gdb) list 72   List lines 67-76
                (the 10 lines around 72)
(gdb) list hello_world.c:main
                List the function main()
                in the code unit hello_world.c
```

Printing the values of expressions

You can print the values of expressions involving variables based on the state of the execution of the process. You can also specify the formatting of those expressions, such as asking for hexadecimal or octal values.

```
print[/FMT] EXPRESSION
```

Printing the values of expressions

The FMT can be ‘o’ for octal, ‘x’ for hexadecimal, ‘d’ for signed decimal, ‘f’ for float, ‘u’ for unsigned decimal, ‘t’ for binary, and ‘a’ for address. If not EXPRESSION is given, the previous one is used.

Example print commands

```
print i      print the value of i
p a[i]       print the value of a[i]
p/t a[i]     print the value of a[i] in binary
p a[i]-x     print the value of a[i]-x
print a      print the values in array a
p *p         print the value pointed to by pointer p
```

Displaying the value of variables

The display command is very similar to the print command, but the value displayed after each step or continue command.

```
display[/FMT] EXPRESSION
```

You can use “undisplay” to stop displaying expressions.

Showing the registers and memory

You can use “info reg” to show the registers, and use the “x” command to display memory:

```
(gdb) info reg              display most useful registers
(gdb) i r
(gdb) i all                 display all registers
(gdb) x/x 0xff00ff0fff00 display memory
(gdb) x/24x do_preload   show 24 bytes in hexadecimal of
                            the function do_preload
```

Stack traces

You can print a trace of the activation records (aka frames) of the stack of functions.

The trace shows the names of the functions, the values of the arguments passed to each, and the line last executed in that routine.

Stack traces

The general form is

`where [N]`

If N is positive, then only the last N activation records are shown.

If N is negative, then only the first N activation records are shown.

Moving around the stack

```
(gdb) up        Go up the stack
(gdb) down      Go down the stack
(gdb) frame 3   Specify stack frame 3
```

The “frame” command is particularly useful for heavily recursive code.

(frames.c)

Changing state

You can modify the values of variables while executing; this can, for instance, save you from making code changes just for the sake of debugging.

For instance:

```
set i = 10
set a[i] = 4
```

Making impromptu calls to functions

You can directly invoke a function from the gdb prompt. This can be very useful to call debugging routines that print the values of complex structures that might be difficult to parse with raw gdb print commands:

```
(gdb) call FUNCTION(ARGS,...)
```

Other useful features

One of the most useful things that you can do is to simply run a program that is segfaulting and see where the problem is occurring. Or if you have a core file from a segfaulted program, you can specify to read its states with

```
gdb PROGRAMNAME COREFILE
```

Other useful features

You can also just CTRL-C when you are in an endless loop and find out exactly where the infinite loop is occurring.

Command shortcuts

You can create and use aliases:

```
(gdb) alias x1=print
(gdb) x1 arg1
```