IBM

developerWorks

# Linux software debugging with GDB

## An introduction to the GNU debugger

David Seager
CICS/390 Development
IBM Hursley

01 February 2001

Most flavours of Linux come with the GNU debugger, or gdb to the shell. Gdb lets you see the internal structure of a program, print out variable values, set breakpoints and single step through source code. It makes an extremely powerful tool for fixing problems in program code. In this article I'll try to show how cool and useful gdb is.

## Compiling

Before you can get started, the program you want to debug has to be compiled with debugging information in it. This is so gdb can work out the variables, lines and functions being used. To do this, compile your program under gcc (or g++) with an extra '-g' option:

gcc -g eg.c -o eg

## Running gdb

Gdb is run from the shell with the command 'gdb' with the program name as a parameter, for example 'gdb eg', or you can use the file command once inside gdb to load a program for debugging, for example 'file eg'. Both of these assume you execute the commands from the same directory as the program. Once loaded, the program can be started with the gdb command 'run'.

## An example debugging session

If nothing is wrong your program will execute to completion, at which point gdb will get control back. But what if something does go wrong? In this case gdb will take control and interrupt the program, allowing you to examine the state of everything and hopefully find out why. To provoke this scenario we'll use an example program:

```
#include
int wib(int no1, int no2)
{
  int result, diff;
  diff = no1 - no2;
  result = no1 / diff;
  return result;
}
int main(int argc, char *argv[])
{
  int value, div, result, i, total;
  value = 10;
  div = 6;
  total = 0;
  for(i = 0; i < 10; i++)
  {
    result = wib(value, div);
    total += result;
    div++;
    value--;
  }
  printf("%d wibed by %d equals %d\n", value, div, total);
  return 0;
}
```

This program runs around a for loop 10 times, calculating a cumulative value using the 'wib()' function and finally printing out the result.

Enter it into your favorite text editor with the same line spacings, save as 'eg1.c', compile with 'gcc -g eg1.c -o eg1' and start gdb with 'gdb eg1'. Rupning the program using 'run' will result in a message something like:

```
Program received signal SIGFPE, Arithmetic exception.
0x80483ea in wib (no1=8, no2=8) at eg1.c:7
7         result = no1 / diff;
(gdb)
```

Gdb indicates that the program gets an arithmetic exception at line 7 and usefully prints out the line and the values of the arguments to the wib() function. To see the source code around line 7 use the command 'list', which usually prints 10 lines. Typing 'list' again (or pressing return which repeats the last command) will list the next 10 lines of the program. From the gdb message something is going wrong with the divide at line 7 where the program divides the variable "no1" by "diff".

To see the values of variables the gdb 'print' command is used with the variable name. We can see what "no1" and "diff" are equal to by typing 'print no1' and 'print diff', resulting in:

```
(gdb) print no1
$5 = 8
(gdb) print diff
$2 = 0
```

Gdb indicates that "no1" equals 8 and "diff" equals 0. From these values and line 7 we can deduce that the arithmetic exception is due to a divide by zero. The listing shows the variable "diff" being calculated on line 6, which we can re-evaluate by supplying the "diff" expression to print as 'print no1 - no2'. Gdb told us that the arguments to the wib function were both equal to 8 so we might wish to examine the main() function which calls wib() to see when this happens. In the meantime to allow our program to die naturally we tell gdb to carry on execution with the 'continue' command:

```
(gdb) continue
Continuing.
Program terminated with signal SIGFPE, Arithmetic exception.
The program no longer exists.
```

## Using breakpoints

To see what's going on in main() we can set a breakpoint at a particular line or on a function in the program code so gdb will interrupt execution when it is reached. We could set a breakpoint when the main() function is entered with the command 'break main', or specify any other function name we were interested in. For our purposes however we'll break just before the wib() function is called. Typing 'list main' will print the source listing starting around the main() function and pressing return again will reveal the wib() function call is on line 21. To set a breakpoint there we type 'break 21'. Gdb will issue the response:

```
(gdb) break 21
Breakpoint 1 at 0x8048428: file eg1.c, line 21.
```

to show that it has set breakpoint 1 at the line we requested. The 'run' command will rerun the program from the beginning until gdb breaks. When this happens gdb will generate a message showing which breakpoint it broke on and where the program was:

```
Breakpoint 1, main (argc=1, argv=0xbffff954) at eg1.c:21
21          result = wib(value, div);
```

Issuing 'print value' and 'print div' will show that the variables are 10 and 6 for this first wib() call and 'print i' will show zero. Happily gdb will show the value of all local variables and save a lot of typing with the 'info locals' command.

From the previous investigation the problem occurs when "value" and "div" are equal, so type 'continue' to resume execution until breakpoint 1 is next reached. For this iteration 'info locals' shows value=9 and div=7.

Rather than continuing again we can single step through the program to see how "value" and "div" are being changed using the command 'next'. Gdb will respond with:

```
(gdb) next
22          total += result;
```

pressing return a couple more times will show an addition and subtraction:

```
(gdb)
23          div++;
(gdb)
24          value--;
```

and another two returns will get us to line 21, ready for the wib() call. 'info locals' will show that now "div" equals "value", spelling forthcoming trouble. For interest's sake we can follow execution into the wib() function to see the divide error again by issuing the 'step' command (as opposed to 'next' which steps over function calls) followed by a 'next' to get to the "result" calculation.

Now that we're finished debugging, gdb can be exited with the 'quit' command. Because the program is still running and this action will terminate it, gdb will prompt for confirmation.

## More breakpoints and watchpoints

In the previous example we set a breakpoint at line 21 because we were interested in when "value" equaled "div" before the wib() function was called. We had to continue program execution twice to get to this point, however by setting a condition on the breakpoint we can make gdb halt only when "value" actually equals "div". To set the condition when defining the breakpoint we can specify "break <line number> if <conditional expression>". Load eg1 into gdb again and type:

```
(gdb) break 21 if value==div
Breakpoint 1 at 0x8048428: file eg1.c, line 21.
```

If a breakpoint such as number 1 was already defined at line 21 we could use the 'condition' command instead to set the condition on the breakpoint:

```
(gdb) condition 1 value==div
```

Running eg1.c with 'run' gdb will break when "value" equals "div", avoiding having to 'continue' manually until they are equal. Breakpoint conditions can be any valid C expression when debugging C programs, indeed any valid expression in the language your program is using. The variables specified in the condition must be in scope at whatever line you set the breakpoint on, otherwise the expression wouldn't make sense!

Breakpoints can be set to unconditional using the 'condition' command specifying a breakpoint number without an expression, for example 'condition 1' sets breakpoint 1 to unconditional.

To see what breakpoints are currently defined and their conditions issue the command 'info break':

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048428 in main at eg1.c:21
        stop only if value == div
        breakpoint already hit 1 time
```

Along with any conditions and how many times it's been hit, the breakpoint information specifies whether the breakpoint is enabled in the 'Enb' column. Breakpoints can be disabled using the command 'disable <breakpoint number>', enabled with 'enable <breakpoint number>' or deleted entirely with 'delete <breakpoint number>', for example 'disable 1' prevents breaking on point 1.

If we were more interested in when "value" became equal to "div" we could set a different type of breakpoint called a watch. A watchpoint will break program execution when the specified expression changes value, but it must be set when the variables used in the expression are in scope. To get "value" and "div" in scope we can set a breakpoint on main and run the program, setting our watchpoints when the main() breakpoint is hit. Restart gdb with eg1 and type:

```
(gdb) break main
Breakpoint 1 at 0x8048402: file eg1.c, line 15.
(gdb) run
...
Breakpoint 1, main (argc=1, argv=0xbffff954) at eg1.c:15
15        value = 10;
```

To keep track of when "div" changes we could use 'watch div', but as we want to break when "div" equals "value" type:

```
(gdb) watch div==value
Hardware watchpoint 2: div == value
```

Continuing will result in Gdb breaking when the expression "div==value" changes value from 0 (false) to 1 (true):

```
(gdb) continue
Continuing.
Hardware watchpoint 2: div == value
Old value = 0
New value = 1
main (argc=1, argv=0xbffff954) at eg1.c:19
19        for(i = 0; i < 10; i++)
```

An 'info locals' command will verify that "value" is indeed equal to "div" (8 again).

Defined watchpoints can be listed along with breakpoints with the 'info watch' command (the command is equivalent to 'info break') and watchpoints can be enabled, disabled and deleted using the same syntax as for breakpoints.

## Core files

Running programs under gdb makes for easier bug trapping, but usually a program will die outside of the debugger leaving only a core file. Gdb can load core files and let you examine the state of the program before it died.

Running our example program eg1 outside of gdb will result in a core dump:

```
$ ./eg1
Floating point exception (core dumped)
```

To start gdb with a core file, issue the command 'gdb eg1 core' or 'gdb eg1 -c core' from the shell. Gdb will load up the core file, eg1's program listing, show how the program terminated and present a message very much like we've just run the program under gdb:

```
...
Core was generated by `./eg1'.
Program terminated with signal 8, Floating point exception.
...
#0  0x80483ea in wib (no1=8, no2=8) at eg1.c:7
7         result = no1 / diff;
```

At this point we can issue 'info locals', 'print', 'info args' and 'list' to see the values which caused the divide-by-zero. The command 'info variables' will print out the values of all program variables, but will take a long time because gdb prints variables from the C library as well as our program code. In order to more easily find out what happened in the function which called wib() we can use gdb's stack commands.

## Stack traces

The program "call stack" is a list of functions which led up to the current one. Each function and its variables are assigned a "frame" with the most recently called function in frame 0 (the "bottom" frame). To print the stack, issue the command 'bt' (short for 'backtrace'):

```
(gdb) bt
#0  0x80483ea in wib (no1=8, no2=8) at eg1.c:7
#1  0x8048435 in main (argc=1, argv=0xbffff9c4) at eg1.c:21
```

This shows that the function wib() was called from main() at line 21 (a quick 'list 21' will confirm this), that wib() is in frame 0 and main() in frame 1. Because wib() is in frame 0 that's the function the program was executing inside when the arithmetic error occurred.

When you issue the 'info locals' command gdb actually prints out variables local to the current frame, which by default is where the interrupted function is (frame 0). The current frame can be printed with the command 'frame'. To see variables from the main function (which is in frame 1) we can switch to frame 1 by issuing 'frame 1' followed by 'info locals':

```
(gdb) frame 1
#1  0x8048435 in main (argc=1, argv=0xbffff9c4) at eg1.c:21
21            result = wib(value, div);
(gdb) info locals
value = 8
div = 8
result = 4
i = 2
total = 6
```

This shows that the error occurred on the third time through the "for" loop (i equals 2) when "value" equaled "div".

Frames can be switched through either by specifying their number explicitly to the 'frame' command as above or with the command 'up' to move up the stack and 'down' to move down. To get further information about a frame such as its address and the program language you can use the command 'info frame'.

Gdb stack commands work during program execution as well as on core files, so for complicated programs you can trace how the program arrives at functions while it's running.

## Attaching to other processes

In addition to debugging with core files or programs, gdb can attach to an already running process (who's program has debugging information compiled in) and break into it. This is done by specifying the process ID of the program you wish to attach gdb to instead of the core filename. Here's an example program which goes round in a loop and sleeps:

```
#include
int main(int argc, char *argv[])
{
  int i;
  for(i = 0; i < 60; i++)
  {
    sleep(1);
  }
  return 0;
}
```

Compile this with 'gcc -g eg2.c -o eg2' and run it with './eg2 &'. Take note of the process ID which is printed when it starts in the background, in this case 1283:

```
./eg2 &
[3] 1283
```

Start gdb and specify your pid, in my case with 'gdb eg2 1283'. Gdb will look for a core file called "1283" and when it doesn't find it will attach and break into to process 1283, wherever it's running (in this case probably in sleep()):

```
...
/home/seager/gdb/1283: No such file or directory.
Attaching to program: /home/seager/gdb/eg2, Pid 1283
...
0x400a87f1 in __libc_nanosleep () from /lib/libc.so.6
(gdb)
```

At this point all the usual gdb commands can be issued. We can use 'backtrace' to see where we are in relation to main() and what main()'s frame number is, then switch frames there and find out how many times we've been through the "for" loop:

```
(gdb) backtrace
#0  0x400a87f1 in __libc_nanosleep () from /lib/libc.so.6
#1  0x400a877d in __sleep (seconds=1) at ../sysdeps/unix/sysv/linux/sleep.c:78
#2  0x80483ef in main (argc=1, argv=0xbffff9c4) at eg2.c:7
(gdb) frame 2
#2  0x80483ef in main (argc=1, argv=0xbffff9c4) at eg2.c:7
7           sleep(1);
(gdb) print i
$1 = 50
```

When we're done with the program we can leave it to carry on executing with the 'detach' command or kill it with the 'kill' command. We could also attach to eg2 under pid 1283 by first loading the file in with 'file eg2' then issuing the attach command 'attach 1283'.

## Other neat tricks

Gdb allows you to run shell commands without exiting the debugging environment with the shell command, invoked as 'shell [commandline]', useful for making changes to source code whilst debugging.

Finally you can modify the values of variables whilst a program is running using the 'set ' command. Run eg1 again under gdb, set a conditional breakpoint at line 7 (where result is calculated) with the command 'break 7 if diff==0' and run the program. When gdb interrupts execution "diff" can be set to a non-zero value to get the program to run to completion:

```
Breakpoint 1, wib (no1=8, no2=8) at eg1.c:7
7         result = no1 / diff;
(gdb) print diff
$1 = 0
(gdb) set diff=1
(gdb) continue
Continuing.
0 wibed by 16 equals 10
Program exited normally.
```

## Conclusion

The GNU Debugger is a very powerful tool in any programmer's arsenal. I've only covered a small subset of what it can do here, to find out more I'd encourage you to have a read of the GNU Debugger manual pages.

## Resources

- GNU Debugger manual
- Source code for the example debugging session.
- Source code for the *attach* example.

## About the author

### David Seager

David Seager, a software developer with IBM, has been playing with Linux and Web-based applications for over 2 years.