Using gdb

Adobe PDF Format

The GNU debugger, gdb, is your best friend when one of your programs continually crashes and you don't know why. gdb allows you to run your program inside of a controlled environment - you can arbitrarily start and stop execution, watch the values of variables change, trace execution one line at a time, and even analyze the results of a program crash to see what went wrong. gdb includes all the features found in other debugging products, such as those included with Borland's C++ Builder and Microsoft Visual Studio (there's more than a passing resemblance between the Visual Studio debugger and gdb - Microsoft has been known to "borrow" code now and then). gdb is a text-based debugger, but there are graphical interfaces to it for those that insist on point-and-click.

This document walks through a typical gdb debugging session. The code being debugged is very simplistic, but the same commands and principles apply to much more complicated programs.

The complete source code for the program being debugged is included at the end of this document, if you want to place it on your system and follow along with the demonstration. (Make sure you have gdb installed on your system first).

Assuming that we've just finished writing our program named `example.c`, we want to compile and run it to make sure it works.

```
hemicuda demo> gcc -o example example.c
hemicuda demo> ./example

Initial matrix contents:
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
Segmentation Fault(coredump)
```

Whoops! Something in the program is definitely broken, and the coredump message indicates that the operating system squashed it like a bug. Since it's 5:00am and we haven't slept in 48 hours, a quick glance over the source code doesn't shed any light on what's causing the problem.

Before we can crank up gdb, we have to recompile the program with debugging symbols enabled and optimizations disabled. The `-g` option tells most compilers to include symbolic information about the program in the executable file - this makes it easier for the programmer to figure out what's going on since `value` makes more sense than `0x149387602`. The option `-O0` (that's the capital letter oh followed by the number zero) tells most compilers not to perform any sort of optimization. This is important, because most compiler optimizations are extremely complex. Unless you helped write the compiler, trying to debug an optimized program is

impossible. Execution appears to hop from line to line in no logical order, the values of variables don't make sense at various locations, and so on.

```
hemicuda demo> gcc -g -O0 -o example example.c
```

Now that we have a version of our program suitable for debugging, we can open it inside of gdb.

```
hemicuda demo> gdb example

GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.9"...
(gdb)
```

The (gdb) prompt tells us that gdb is ready and awaiting commands. Typing help at the prompt will give you a list of available commands.

The first thing we want to do is run the program inside of the debugger to see exactly where it dies.

```
(gdb) run
Starting program: /lamont/development/demo/example
Initial matrix contents:
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100

Program received signal SIGSEGV, Segmentation fault.
0x10b4c in transformMatrix (matrix=0xeffffb40) at example.c:58
58            matrix[i][j] = bimod(matrix[i][j]);
```

According to gdb, the error is occurring on line 58. This line is inside the transformMatrix() function and is a call to the bimod() function. The bimod() function is so simple that we're pretty sure it's not the problem, so something in transformMatrix() must be causing our error.

Our next step is to use the list command to print the code surrounding line 58 so we can get an idea of where we are in the program.

```
(gdb) list
53      {
54        int i, j;
```

```
55
56        for (i = 0; i < 100; i++)
57          for (j = 0; j < 100; j++)
58            matrix[i][j] = bimod(matrix[i][j]);
59      }
60
61
62
```

Setting a breakpoint in the `transformMatrix()` function will halt the program's execution and let us step through the code one line at a time. A breakpoint can be set either by specifying the name of the function or a line number - the example below uses the name of the function, but the command `break 56` would do the same thing.

```
(gdb) break transformMatrix
Breakpoint 1 at 0x10aec: file example.c, line 56.
```

Now if we run the program again, it will stop at the beginning of `transformMatrix()` and wait for further instructions.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /lamont/development/demo/example
Initial matrix contents:
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100

Breakpoint 1, transformMatrix (matrix=0xeffffb28) at example.c:56
56        for (i = 0; i < 100; i++)
```

Now we can run through code one line at a time using the `next` command.

```
(gdb) next
57          for (j = 0; j < 100; j++)
(gdb) next
58            matrix[i][j] = bimod(matrix[i][j]);
```

Even though we're fairly confident that the `bimod()` function isn't the problem, we can use the `step` command to trace through it instead of simply executing it like `next` would just to be sure.

```
(gdb) step
bimod (value=1) at example.c:65
65        if (value % 2 == 0)
(gdb) next
```

```
68          return(0);
(gdb) next
69      }
(gdb) next
transformMatrix (matrix=0xeffffb28) at example.c:57
57          for (j = 0; j < 100; j++)
```

When we step into the function, gdb prints the function name, the values that are passed to it as variables, and the line number that it starts on. The next command lets us walk through bimod(), and puts us back in transformMatrix() when bimod() finishes.

Now that we've walked through the functions once, let's clear the breakpoint and continue executing the program. We know the program is still going to stop executing when the problem occurs.

```
(gdb) clear transformMatrix
Deleted breakpoint 1
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x10b4c in transformMatrix (matrix=0xeffffb28) at example.c:58
58            matrix[i][j] = bimod(matrix[i][j]);
```

Sometimes using the print command to examine the values of variables can provide a clue to what went wrong. Let's look at the three variables used in the transformMatrix() function - i, j , and matrix.

```
(gdb) print matrix
$1 = (int (*)[10]) 0xeffffb28
(gdb) print i
$2 = 22
(gdb) print j
$3 = 90
```

As the Crocodile Hunter would say, "Crikey!"

The value of matrix is what we would expect, but the values for i and j are way out of the range they should be in. They're used to reference elements in a [10][10] matrix - no wonder there was a problem when they tried to reference the element at [22][90]!

A quick look at the loops that set the values of i and j tell the story - a programmer wasn't paying attention and added an extra zero to the upper bound of the loop. After exiting gdb, the loop conditions can be changed to fix the problem.

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
hemicuda demo>
```

Source Code

```
#include <stdlib.h>
```

```
#include <stdio.h>

//function prototypes
void transformMatrix(int matrix[10][10]);
int bimod(int value);


int main()
{
  int i, j;
  int matrix[10][10];

  //load values into matrix
  for (i = 0; i < 10; i++)
  {
   for (j = 0; j < 10; j++)
    {
      matrix[i][j] = (i+1) * (j+1);
    }
  }

  //display initial contents of matrix
  printf("Initial matrix contents:\n");
  for (i = 0; i < 10; i++)
  {
    for (j = 0; j < 10; j++)
    {
      printf("%4i", matrix[i][j]);
    }
    printf("\n");
  }

  transformMatrix(matrix);

  //display transformed contents of matrix
  printf("\nTransformed matrix contents:\n");
  for (i = 0; i < 10; i++)
  {
    for (j = 0; j < 10; j++)
    {
      printf("%4i", matrix[i][j]);
    }
    printf("\n");
  }

  exit(0);
}


void transformMatrix(int matrix[10][10])
{
  int i, j;

  for (i = 0; i < 100; i++)
    for (j = 0; j < 100; j++)
      matrix[i][j] = bimod(matrix[i][j]);
}


int bimod(int value)
{
  if (value % 2 == 0)
    return(1);
```

```
  else
    return(0);
}
```

Michael's Homepage WKU-Linux