Introduction to debugging with GDB

Spring, 2024

Summary

This document is meant to be used as a supplement for an optional practice session focused on using gdb for reasoning about code written in C or assembly.

Corequisites

- Computer Architecture, Operating Systems and Networks

Class Objectives

1. Understand gdb workflow.
2. Know core gdb commands.
3. Be able to debug C with gdb.

Introduction

In your favorite IDEs, you may have used a graphical debugger; these debuggers were built into the program you used to write your code, and allowed you to set breakpoints, step through your code, and see variable values, among other features. Here, the debugger we are using is a separate program from your text editor, called gdb (the "GNU Debugger"). It is a command-line debugger, meaning that you interact with it on the command line using text-based commands. But it shares many similarities with debuggers you might have already used; it also allows you to set breakpoints, step through your code, and see variable values. We recommend familiarizing yourself with how to use gdb as it is widely encountered in UNIX systems. gdb allows to debug programs in a few languages, including programs written in assembly. In this document, we are going to focus on C. However, most of the material also applies to assembly.

This page will list the basic gdb commands. However, it takes practice to become proficient in using the tool, and gdb is a large program that has a tremendous number of features. See the bottom of the page for more resources to help you master gdb.

Getting started

As a running example, we are going to use the following buggy program. Its intended behavior is:

- It is called with a single command line argument n.
- The program allocates an array of integers of size n, and initializes it with a sequence of ascending numbers, starting with 0.
- The program queries its user for a number in-between 0 and n-1, and outputs the corresponding value, stored in the array.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int tester (int *c, int k) {
  printf("x[%d] = %d\n", k, c[k]);
}

void process_input(int k, int n) {
  printf ("Enter integer in 0..n-1: ");
  if (scanf("%d", k) != 1) {
    fprintf(stderr, "Incorrect input!\n");
    exit(-1);
  }
  if (k < 0 || k > n) {
    fprintf(stderr, "Incorrect input!\n");
    exit(-1);
  }
}

int main (int argc, char **argv) {
  int n, k;
  int *x;
  if (argc != 2) {
    fprintf(stderr, "Usage: <size>\n");
    exit(-1);
  }
  n = atoi(argv[1]);
  x = malloc(sizeof(int) * n);
  for (int i = 0; i <= n; ++i) {
    x[i] = i;
  }
  process_input(k, n);
  tester(x, k);
}
```

Listing 1: bug.c

The implementation contains some bugs though, which we want to identify and fix.
Firstly, let us compile it and see what goes wrong.

```
$ gcc bug.c -o bug
$ ./bug 100
```

```
Enter integer in 0..n-1: 10
Segmentation fault (core dumped)
```

The first bug is rather easy to fix, and `gcc` itself helps us with a warning:

```
bug.c: In function "process_input":
bug.c:11:15: warning: format "%d" expects argument of type "int␣*", but argument 2 has type "int"
    [-Wformat=]
  11 | if (scanf("%d", k) != 1) {
     | ~^ ~
     | | |
     | | int
     | int *
```

However, for the sake of practice, let us see how we can find it with `gdb`.

Compiling for gdb: `gcc` does not automatically put debugging information into the executable program, but we can include it with `-g -Og` flags that give gdb information about our programs so we can use the debugger efficiently.

Running gdb: `gdb` takes as its argument the executable file that you want to debug. This is not the `.c` file or the `.o` file, instead it is the name of the compiled program.

Layouts: `gdb` has different modes of presentation.

- `layout asm` shows a window with assembly line corresponding to your program.
- `layout regs` shows a window with register values, and updates it while the program is running.
- `layout asm` shows a window with the original source code together with line numbers, breakpoints, and the current stopping point.

Let us compile the program and start debugging!

```
$ gcc bug.c -g -Og -o bug
$ gdb ./bug
...
For help, type "help".
Type "apropos␣word" to search for commands related to "word"...
Reading symbols from ./bug...
(gdb)
```

Breakpoints and execution

Normally, your program only stops when it exits. Breakpoints allow you to pause your program's execution wherever you want, be it at a function call or a particular line of code, and examine the program state.

Before you start running your program, you want to set up your breakpoints. The `break` command (shorthand: `b`) allows you to do so.

To set a breakpoint at the beginning of the function named main:

```
(gdb) break main
Breakpoint 1 at 0x40123e: file bug.c, line 21.
```

To set a breakpoint at line 11 in bug.c:

```
(gdb) b bug.c:11
Breakpoint 2 at 0x4011bd: file bug.c, line 11.
```

If there is only once source file, you do not need to include the filename.

```
(gdb) b 33
Breakpoint 3 at 0x4012ae: file bug.c, line 33.
```

Each breakpoint you create is assigned a sequentially increasing number (the first breakpoint is 1, the second 2, etc.).

If you want to delete a breakpoint, just use the `delete` command (shorthand: `d`) and specify the breakpoint number to delete.

To delete the breakpoint numbered 2:

```
(gdb) delete 2
```

If you lose track of your breakpoints, or you want to see their numbers again, the `info break` command lets you know the breakpoint numbers:

```
(gdb) info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x000000000040123e in main at bug.c:21
3 breakpoint keep y 0x00000000004012ae in main at bug.c:33
```

Now we can control the execution of our program. To run it with command line arguments, we can use `run` command, to proceed past breakpoints we can use `continue` (or `c`).

`run` will start (or restart) the program from the beginning and continue execution until a breakpoint is hit or the program exits. `start` will start (or restart) the program and stop execution at the beginning of the main function.

Once stopped at a breakpoint, you have choices for how to resume execution. `continue` (shorthand: `c`) will resume execution until the next breakpoint is hit or the program exits. `finish` will run until the current function call completes and stop there. You can single-step through the C source using the `next` (shorthand: `n`) or `step` (shorthand: `s`) commands, both of which execute a line and stop. The difference between these two is that if the line to be executed is a function call, `next` executes the entire function, but `step` goes into the function implementation and stops at the first line.

```
(gdb) run 100
Starting program: .../bug 100
...
Breakpoint 1, main (argc=2, argv=0x7fffffffa5e8) at bug.c:21
21 int main (int argc, char **argv) {
(gdb) c
Continuing.
```

```
Breakpoint 3, main (argc=<optimized out>, argv=<optimized out>) at bug.c:33
33 process_input(k, n);
(gdb) c
Continuing.
Enter integer in 0..n-1: 10

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e3d28a in __vfscanf_internal ()
  from .../lib/libc.so.6
```

This time, the program also crashed with a segfault. However, this time we got more information.

Backtracing

Easily one of the most immediately useful things about `gdb` is its ability to give you a backtrace (or a "stack trace") of your program's execution at any given point. This works especially well for locating things like crashes ("segfaults"). In our case, we got the following information:

```
Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e3d28a in __vfscanf_internal ()
  from .../lib/libc.so.6
```

Not only is this information vastly more useful than the terse "Segmentation fault" error that you get outside of `gdb`, you can use the `backtrace` command to get a full stack trace of the program's execution when the error occurred:

```
0x00007ffff7e3d28a in __vfscanf_internal () from .../lib/libc.so.6 #1
0x00007ffff7e2de62 in __isoc99_scanf () from .../lib/libc.so.6 #2
0x00000000004011d0 in process_input (k=k@entry=0, n=100) at bug.c:11
#3 0x00000000004012bc in main (argc=<optimized out>, argv=<optimized
out>) at bug.c:33
```

Each line represents a stack frame (ie. a function call). Frame #0 is where the error occurred, during the call to `process_input`. The hexadecimal number 0x00007ffff7e3d28a is the address of the instruction that caused the segfault. Finally, you see that the error occurred from the code in `bug.c`, on lines 11 and 33. All of this is helpful information to go on if you're trying to debug the segfault.

Variables

You're likely to want to check into the values of certain key variables at the time of the problem. The `print` command (shorthand: `p`) is perfect for this. To print out the value of variables such as `n` and `k` at the second breakpoint, restart the program by restarting with `run`, go to the second breakpoint, and do the following:

```
(gdb) print n
$1 = 100
(gdb) print k
```

```
$2 = 0
```

You can also use `print` to evaluate expressions, make function calls, reassign variables, and more.

Changes `n`:

```
print n = 90
$3 = 90
```

The following commands are handy for quickly printing out a group of variables in a particular function:

`info args` prints out the arguments (parameters) to the current function you're in (and `info locals` prints out the local variables of the current function):

```
0 n = 90
```

In addition, we can create our own variables by using `set`.

```
(gdb) set $foo = (int *)malloc(sizeof(int))
(gdb) p $foo
$4 = (int *) 0x405440
(gdb) p k = $foo
$5 = 4215872
```

Now we can check if passing a valid pointer to integer solves our previous problem.

```
(gdb) n
Enter integer in 0..n-1: 10
15 if (k < 0 || k > n) {
```

It does! We fixed a segfault.

Now, given our analysis, we can fix the program.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int tester (int *c, int k) {
 printf("x[%d] = %d\n", k, c[k]);
}

void process_input(int *k, int n) {
 printf ("Enter integer in 0..n-1: ");
 if (scanf("%d", k) != 1) {
  fprintf(stderr, "Incorrect input!\n");
  exit(-1);
 }
 if (*k < 0 || *k > n) {
  fprintf(stderr, "Incorrect input!\n");
  exit(-1);
 }
}

int main (int argc, char **argv) {
 int n, k;
```

```
  int *x;
  if (argc != 2) {
    fprintf(stderr, "Usage: <size>\n");
    exit(-1);
  }
  n = atoi(argv[1]);
  x = malloc(sizeof(int) * n);
  for (int i = 0; i <= n; ++i) {
    x[i] = i;
  }
  process_input(&k, n);
  tester(x, k);
}
```

`print` also allows to output arrays. E.g., we can fix an off-by-one bug in the `main` function, if we place a breakpoint after the loop, and do the following:

```
(gdb) print *x@100
$1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99}
(gdb) print *x@101
$2 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100}
(gdb) print *x@102
$3 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66,
67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88,
89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 0}
```

We can see that we go over allocated boundaries of the array.

Conditional breakpoints:

You can set breakpoints to only trigger when certain conditions in your code are true. For instance, say you have the following loop in your code:

```
for (int i = 0; i <= n; ++i) {
  ...
}
```

If you wanted to step through the code inside the loop just the last time the loop executed, with a normal loop you may have to skip over many program breaks before you get to the part you want to examine. However, gdb lets you add an optional condition (in C code syntax) for when the breakpoint should be stopped at:

```
(gdb) b 31
(gdb) run 100
```

```
(gdb) condition 1 i == 99
(gdb) c
(gdb) p i
$1 = 99
```

Now this breakpoint will only be hit the last time around the loop! You can even use local variables in your expression, as shown above with i. Experiment to see what other useful conditions you might use.

Exercise

Figure out what's causing that segmentation fault.

Start `gdb` on the program. We recommend setting a breakpoint in the `ll_equal` function. When the debugger stops at the breakpoint, try stepping through the program to see if you can figure out what's causing the error.

Hint 1. Run `backtrace` after the program crashes in `gdb`. It will output the program trace.

Hint 2. Pay attention to the values of the pointers a and b in the function (by printing them).

Hint 3. Look at the main function to see the structure of the nodes and what is being passed into `ll_equal`.

```c
#include <stdio.h>

typedef struct node_st {
  int val;
  struct node_st* next;
} node_t;

/* FIXME */
int ll_equal(const node_t *a, const node_t *b) {
  while (a != NULL) {
    if (a->val != b->val)
      return 0;
    a = a->next;
    b = b->next;
  }
  return a == b;
}

int main(int argc, char **argv) {
  node_t nodes[10];

  for (int i = 0; i < 10; i++) {
    nodes[i].val = 0;
    nodes[i].next = NULL;
  }

  nodes[0].next = &nodes[1];
  nodes[1].next = &nodes[2];
  nodes[2].next = &nodes[3];
  printf("equal test 1 result = %d\n", ll_equal(&nodes[0], &nodes[0]));
  printf("equal test 2 result = %d\n", ll_equal(&nodes[0], &nodes[2]));

  return 0;
}
```

Extra materials

- Debugging with GDB
- Refcard