gdb - An Introduction

Debugging *is* programming. You will spend more of your time debugging than actually writing code. Even very experienced programmers spend a significant time debugging. Furthermore, even if you write rock solid code, chances are you are going to be using libraries and system calls written by someone else. So, you can also end up debugging problems surrounding the interfaces between your code and other code.

The GNU debugger (`gdb`) is a very popular command-line tool for stepping through and analyzing C code. It can also be used with other languages such as C++ and assembly. For our purposes, we will also use `gdb` to debug the Pintos operating system. Pintos is setup to allow for *remote* debugging of the Pintos kernel. Using `gdb` with Pintos will be invaluable for the remaining projects.

The basic idea is that you first start `gdb` by giving it the name of the program (executable) that you want to debug (debugging a kernel will be a little bit different). Once inside gdb, you can set breakpoints at arbitrary locations in the source code. Then you can run your program until the code hits a breakpoint or until the program causes an exception (like a segmentation violation, also known as dereferencing a bad pointer).

The `gdb` program is most useful in finding out how a program is executing by watching the program flow and by inspecting the values of variables at run time. Although `gdb` is very sophisticated, you only need to learn a small subset of the commands in order to make it useful. In linux, there are some GUI debuggers based on `gdb`, such as `ddd`. To get more information on `gdb` type `info gdb`.

gdb - Summary

- `gdb` is an interactive, source level, command-line debugger
- Prepare a binary at compile time using the `-g` option: `gcc -g -o foo foo.c`
- `gdb` can be used to get *breakpoints*. While running your program from within `gdb`, you program will stop at each break point so you can inspect the values of important variables.
- `gdb` allows you to *step* through your program one statement at a time.
- `gdb` can be used to find out were a *segmentation fault* occurs.
- `gdb` allows you to show a *backtrace* of the call stack. This allows you to see the chain of functions calls that leads to the current statement.

gdb command summary

- **r** (run) – start the program to be debugged
- **b line-number** (breakpoint) – set a breakpoint at the specified line number. Use **b function-name** to break at a function.
- **info b** – show all your breakpoints
- **d** – delete all breakpoints
- **d number** – delete a specific breakpoint
- **c** (continue) – continue program execution after stopping at a breakpoint or for CTRL-C.
- **n** (next) – execute the next statement (do not go into functions)
- **s** (step) – execute the next statement (go into functions)

- **p var** (print) – print the value of a variable
- **bt** (backtrace) – show the function call stack
- **q** (quit) – quit gdb

Preparing a program for debugging

In order to debug a program with `gdb` you need to compile your source code (all source modules that you want to debug) with the `-g` option:

```
gcc -g -o foo foo.c
```

Starting gdb

Simply start `gdb` with the name of your executable:

```
$ gcc -g -o foo foo.c
$ gdb foo
GNU gdb 5.1.1
Copyright 2002 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 "i386-suse-linux"...
(gdb)
```

To start your program within `gdb`, simply use the "r" command (for run):

```
(gdb) r
```

Optionally, you can put command arguments that you want to send to your program after the "r" command:

```
(gdb) r arg1 arg2 ... argN
```

Example 1 - setting breakpoints and examining variables

Here is a program that is supposed to add 2 to the variable j in every iteration of the for-loop:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i, j = 0;
        for (i = 0; i < 100; i++); /* <-- unwanted semicolon */
                j += 2;
        printf("The value of j is: %d\n", j);
        return 0;
}
```

Compiling and running the program reveals that it is not adding 2 to j 100 times:

```
$ gcc -g -o ex1 ex1.c
$ ./ex1
The value of j is: 2
```

Because we compiled `ex1.c` with the `-g` option we can run gdb on the `ex1` executable.

```
$ gdb ex1
GNU gdb 5.1.1
Copyright 2002 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 "i386-suse-linux"...
(gdb) l
1       #include <stdio.h>
2
3       int main(int argc, char *argv[])
4       {
5               int i, j = 0;
6               for (i = 0; i < 100; i++); /* <-- unwanted semicolon */
7                       j += 2;
8               printf("The value of j is: %d\n", j);
9               return 0;
10      }
(gdb)
```

Using the **b** command we can set a breakpoint at the beginning of the `main` function.

```
(gdb) b main
Breakpoint 1 at 0x8048456: file ex1.c, line 5.
```

Now we can begin executing the program using the the **r** command:

```
(gdb) r
Starting program: /home/benson/tch/usf/cs326/2002F/handouts/03-gdb/src/ex1

Breakpoint 1, main (argc=1, argv=0x3ffff184) at ex1.c:5
5               int i, j = 0;
```

The **n** command executes the next statement in the program:

```
(gdb) n
6               for (i = 0; i < 100; i++); /* <-- unwanted semicolon */
(gdb) n
7                       j += 2;
(gdb)
```

Note that the line displayed after typing **n** is the next line of the program to be executed. Up to this point we have executed the for-loop, but have not executed `j += 2`. Now we can use the `print` command to view the values of i and j:

```
(gdb) p i
$1 = 100
(gdb) p j
$2 = 0
```

After executing the for-loop, j should be 200. It seems that j is not getting incremented. Close inspection of the for-loop reveals that we have a semi-colon where we do not want one. If we remove the semi-colon, j will get incremented properly. To end the gdb session we can use the `quit` command:

```
(gdb) q
The program is running.  Exit anyway? (y or n) y
```

Example 2 - finding a segmentation fault

You can also use `gdb` to find the location of a segfault.

Consider the following program:

```
#include<stdio.h>

int main(int argc, char *argv[])
{
        int a[10];
        int *a1;


        a1 = a;

        a1 = 0;

        a1[0] = 1;

        return 0;
}
```

Let's run it under `gdb`:

```
[2785]orb: gdb segfault
GNU gdb 5.1.1
Copyright 2002 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 "i386-suse-linux"...
(gdb) r
Starting program: /home/benson/tch/usf/cs326/2002F/handouts/03-gdb/src/segfault

Program received signal SIGSEGV, Segmentation fault.
0x08048426 in main (argc=1, argv=0x3ffff174) at segfault.c:13
13              a1[0] = 1;
(gdb)
```

Notice how `gdb` shows exactly where the segfault occurs.

Example 3 - examining memory

Consider the following program:

```
#include<stdio.h>
#include<stdint.h>
#include<string.h>

struct foo {
        char name[32];
        int age;
        int weight;
};

uint8_t memory[32768];


void print_foo(struct foo *fp)
{
        printf("Name: %s, Age: %d, Weight: %d\n",
                fp->name, fp->age, fp->weight);
}

int main(int argc, char *argv[])
{
        struct foo *foop;

        foop = (struct foo *) &memory[0];

        strcpy(foop->name, "Greg");
        foop->age = 29;
        foop->weight = 150;

        foop = (struct foo *) &memory[(sizeof(struct foo))];

        strcpy(foop->name, "Tony");
        foop->age = 20;
        foop->weight = 200;

        print_foo((struct foo *) &memory[0]);
        print_foo((struct foo *) &memory[(sizeof(struct foo))]);

        return 0;
}
```

We can use gdb to look at structs in memory.

```
$ gdb memory
GNU gdb 6.1-20040303 (Apple version gdb-384) (Mon Mar 21 00:05:26 GMT 2005)
Copyright 2004 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 "powerpc-apple-darwin"
...Reading symbols for shared libraries ... done

(gdb) l
warning: Source file is more recent than executable.

15      {
16                      printf("Name: %s, Age: %d, Weight: %d\n",
17                                      fp->name, fp->age, fp->weight);
18      }
19
20      int main(int argc, char *argv[])
21      {
22                      struct foo *foop;
23
24                      foop = (struct foo *) &memory[0];
(gdb) l
25
26                      strcpy(foop->name, "Greg");
27                      foop->age = 29;
28                      foop->weight = 150;
29
30                      foop = (struct foo *) &memory[(sizeof(struct foo))];
31
32                      strcpy(foop->name, "Tony");
33                      foop->age = 20;
34                      foop->weight = 200;
(gdb) l
35
36                      print_foo((struct foo *) &memory[0]);
37                      print_foo((struct foo *) &memory[(sizeof(struct foo))]);
38
39              return 0;
40      }
(gdb) b 36
Breakpoint 1 at 0x2ac8: file memory.c, line 36.
(gdb) r
Starting program: /Users/benson/Share/Teaching/cs326/handouts/01-gdb/src/memory
Reading symbols for shared libraries . done

Breakpoint 1, main (argc=1, argv=0xbffff740) at memory.c:36
36                      print_foo((struct foo *) &memory[0]);
(gdb) print *((struct foo *) &memory[0])
$1 = {
  name = "Greg", '\0' <repeats 27 times>,
  age = 29,
```

```
  weight = 150
}
(gdb) print sizeof(struct foo)
$2 = 40
(gdb) print *((struct foo *) &memory[40])
$3 = {
  name = "Tony", '\0' <repeats 27 times>,
  age = 20,
  weight = 200
}
(gdb) x/100c memory
0x30d0 <memory>:        71 'G'  114 'r' 101 'e' 103 'g' 0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x30d8 <memory+8>:      0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x30e0 <memory+16>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x30e8 <memory+24>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x30f0 <memory+32>:     0 '\0'  0 '\0'  0 '\0'  29 '\035'       0 '\0'  0 '\0'  0 '\0'  -106 '\226'
0x30f8 <memory+40>:     84 'T'  111 'o' 110 'n' 121 'y' 0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3100 <memory+48>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3108 <memory+56>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3110 <memory+64>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3118 <memory+72>:     0 '\0'  0 '\0'  0 '\0'  20 '\024'       0 '\0'  0 '\0'  0 '\0'  -56 '?'
0x3120 <memory+80>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3128 <memory+88>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'  0 '\0'
0x3130 <memory+96>:     0 '\0'  0 '\0'  0 '\0'  0 '\0'
(gdb)
```