Using GNU's GDB Debugger

Memory Layout And The Stack

By Peter Jay Salzman

Previous: Introduction Next: Debugging With Your Brain

Where Are We Going To Go?

To effectively learn how to use GDB, you must understand frames, which are also called stack frames because they're the frames that comprise the stack. To learn about the stack, we need to learn about the memory layout of an executing program. The discussion will mainly be theoretical, but to keep things interesting we'll conclude the chapter with an example of the stack and stack frames using GDB.

The material learned in this chapter may seem rather theoretical, but it does serve a few very useful purposes:

1. Understanding the stack is absolutely necessary for using a symbolic debugger like GDB.
2. Knowing the memory layout of a process will help us understand what exactly a segmentation fault (or segfault) is, and why they happen (or sometimes, more importantly) don't happen when they should. In brief, segfaults are the most common immediate cause for a program to bomb.
3. A knowledge of a program's memory space can often allow us to figure out the location of well-hidden bugs without the use of `print()` statements, a compiler or even GDB! In the next section, which is a guest written piece by one my friends, Mark Kim, we'll see some real Sherlock Holmes style sleuthing. Mark homes in on a well hidden bug in somewhat lengthy code. It only took him about 5 or 10 minutes, and all he did was look at the program and use his knowledge of how a program's memory space works. It's really impressive!

So without futher ado, let's take a look at how programs are laid out in memory.

Virtual Memory (VM)

Whenever a process is created, the kernel provides a chunk of physical memory which can be located anywhere at all. However, through the magic of virtual memory (VM), the process believes it has all the memory on the computer. You might have heard "virtual memory" in the context of using hard drive space as memory when RAM runs out. That's called virtual memory too, but is largely unrelated to what we're talking about. The VM we're concerned with consists of the following principles:

1. Each process is given physical memory called the process's virtual memory space.
2. A process is unaware of the details of its physical memory (i.e. where it physically resides). All the process knows is how big the chunk is and that its chunk begins at address 0.
3. Each process is unaware of any other chunks of VM belonging to other processes.
4. Even if the process did know about other chunks of VM, it's physically prevented from accessing that memory.

Each time a process wants to read or write to memory, its request must be translated from a VM address to a physical memory address. Conversely, when the kernel needs to access the VM of a process, it must translate a physical memory address into a VM address. There are two major issues with this:

1. Computers constantly access memory, so translations are very common; they must be lighting fast.
2. How can the OS ensure that a process doesn't trample on another process's VM?

The answer to both questions lies in the fact that the OS doesn't manage VM by itself; it gets help from the CPU. Many CPUs contain a device called an MMU: a memory management unit. The MMU and the OS are jointly responsible for managing VM, translating between virtual and physical addresses, enforcing permissions on which processes are allowed to access which memory locations, and enforcing read/write permissions on sections of a VM space, even for the process that owns that space.

It used to be the case that Linux could only be ported to architectures that had an MMU (so Linux wouldn't run on, say, an x286). However, in 1998, Linux was ported to the 68000 which had no MMU. This paved the way for embedded Linux and Linux on devices such as the Palm Pilot.

Exercises

1. Read a short Wikipedia blurb on the MMU
2. Optional: If you want to know more about VM, here's a link. This is much more than you need to know.

Memory Layout

That's how VM works. For the most part, each process's VM space is laid out in a similar and predictable manner:

High Address	Args and env vars	Command line arguments and environment variables
	Stack | V	
	Unused memory	
	^ | Heap	
	Uninitialized Data Segment (bss)	Initialized to zero by `exec`.
	Initialized Data Segment	Read from the program file by `exec`.
Low Address	Text Segment	Read from the program file by `exec`.

- **Text Segment:** The text segment contains the actual code to be executed. It's usually sharable, so multiple instances of a program can share the text segment to lower memory requirements. This segment is usually marked read-only so a program can't modify its own instructions.
- **Initialized Data Segment:** This segment contains global variables which are initialized by the programmer.
- **Uninitialized Data Segment:** Also named "bss" (block started by symbol) which was an operator used by an old assembler. This segment contains uninitialized global variables. All variables in this segment are initialized to 0 or NULL pointers before the program begins to execute.
- **The stack:** The stack is a collection of stack frames which will be described in the next section. When a new frame needs to be added (as a result of a newly called function), the stack grows downward.
- **The heap:** Most dynamic memory, whether requested via C's `malloc()` and friends or C++'s `new` is doled out to the program from the heap. The C library also gets dynamic memory for its own personal workspace from the heap as well. As more memory is requested "on the fly", the heap grows upward.

Given an object file or an executable, you can determine the size of each section (realize we're not talking about memory layout; we're talking about a disk file that will eventually be resident in memory). Given hello_world-1.c, Makefile:

```
// hello_world-1.c

#include <stdio.h>

int main(void)
{
   printf("hello world\n");

   return 0;
}
```

compile it and link it separately with:

```
$ gcc -W -Wall -c hello_world-1.c
$ gcc -o hello_world-1  hello_world-1.o
```

You can use the **`size`** command to list out the size of the various sections:

```
$ size hello_world-1 hello_world-1.o
text   data   bss    dec   hex   filename
 916    256     4   1176   498   hello_world-1
  48      0     0     48    30   hello_world-1.o
```

The data segment is the initialized and uninitialized segments combined. The dec and hex sections are the file size in decimal and hexidecimal format respectively.

You can also get the size of the sections of the object file using "**`objdump -h`**" or "**`objdump -x`**".

```
$ objdump -h hello_world-1.o

hello_world-1.o:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000023  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000058  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000058  2**2
                  ALLOC
  3 .rodata       0000000d  00000000  00000000  00000058  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .note.GNU-stack 00000000  00000000  00000000  00000065  2**0
                  CONTENTS, READONLY
  5 .comment      0000001b  00000000  00000000  00000065  2**0
                  CONTENTS, READONLY
```

Exercises

1. The **`size`** command didn't list a stack or heap segment for hello_world or hello_world.o. Why do you think that is?
2. There are no global variables in hello_world-1.c. Give an explanation for why **`size`** reports that the data and bss segments have zero length for the object file but non-zero length for the executable.
3. **`size`** and **`objdump`** report different sizes for the text segment. Can you guess

where the discrepancy comes from? Hint: How big is the discrepancy? See anything of that length in the source code?

4. Optional: Read this link about object file formats.

Stack Frames And The Stack

You just learned about the memory layout for a process. One section of this memory layout is called the stack, which is a collection of stack frames. Each stack frame represents a function call. As functions are called, the number of stack frames increases, and the stack grows. Conversely, as functions return to their caller, the number of stack frames decreases, and the stack shrinks. In this section, we learn what a stack frame is. A very detailed explanation here, but we'll go over what's important for our purposes.

A program is made up of one or more functions which interact by calling each other. Every time a function is called, an area of memory is set aside, called a stack frame, for the new function call. This area of memory holds some crucial information, like:

1. Storage space for all the automatic variables for the newly called function.
2. The **line number** of the calling function to return to when the called function returns.
3. The arguments, or parameters, of the called function.

Each function call gets its own stack frame. Collectively, all the stack frames make up the **call stack**. We'll use hello_world-2.c for the next example.

```
#include <stdio.h>
void first_function(void);
void second_function(int);

int main(void)
{
   printf("hello world\n");
   first_function();
   printf("goodbye goodbye\n");

   return 0;
}


void first_function(void)
{
   int imidate = 3;
   char broiled = 'c';
   void *where_prohibited = NULL;

   second_function(imidate);
   imidate = 10;
}


```

```
26  void second_function(int a)
27  {
28     int b = a;
29  }
```

When the program starts, there's one stack frame, belonging to **`main()`**. Since **`main()`** has no automatic variables, no parameters, and no function to return to, the stack frame is uninteresting. Here's what the stack looks like just before the call to **`first_function()`** is made.

```
Frame for main()
```

When the call to **`first_function()`** is made, unused stack memory is used to create a frame for **`first_function()`**. It holds four things: storage space for an int, a char, and a void *, and the line to return to within **`main()`**. Here's what the call stack looks like right before the call to **`second_function()`** is made.

```
Frame for main()

Frame for first_function()
     Return to main(), line 9
     Storage space for an int
     Storage space for a char
     Storage space for a void *
```

When the call to **`second_function()`** is made, unused stack memory is used to create a stack frame for **`second_function()`**. The frame holds 3 things: storage space for an int and the current address of execution within **`second_function()`**. Here's what the stack looks like right before **`second_function()`** returns.

```
Frame for main()

Frame for first_function():
     Return to main(), line 9
     Storage space for an int
     Storage space for a char
     Storage space for a void *

Frame for second_function():
     Return to first_function(),
line 22
     Storage space for an int
     Storage for the int
parameter named a
```

When **`second_function()`** returns, its frame is used to determine where to return to (line 22 of **`first_function()`**), then deallocated and returned to stack. Here's what the call stack looks like after **`second_function()`** returns:

```
Frame for main()

Frame for first_function():
     Return to main(), line 9
     Storage space for an int
     Storage space for a char
     Storage space for a void *
```

When **`first_function()`** returns, its frame is used to determine where to return to (line 9 of `main()`), then deallocated and returned to the stack. Here's what the call stack looks like after `first_function()` return:

Frame for main()

And when `main()` returns, the program ends.

Exercises

1. Suppose a program makes 5 function calls. How many frames should be on the stack?
2. We saw that the stack grows linearly downward, and that when a function returns, the last frame on the stack is deallocated and returned to unused memory. Is it possible for a frame somewhere in the middle of the stack to be returned to unused memory? If it did, what would that mean about the running program?
3. Can a `goto()` statement cause frames in the middle of the stack to be deallocated? The answer is no, but why?
4. Can `longjmp()` cause frames in the middle of the stack to be deallocated?

The Symbol Table

A symbol is a variable or a function. A symbol table is exactly what you think: it's a table of variables and functions within an executable. Normally, symbol tables contain only memory addresses of symbols, since computers don't use (or care) what we name variables and functions.

But in order for GDB to be useful to us, it needs to be able to refer to variable and function names, not their addresses. Humans use names like `main()` or `i`. Computers use addresses like `0x804b64d` or `0xbffff784`. To that end, we can compile code with "debugging information" which tells GDB two things:

1. How to associate the address of a symbol with its name in the source code.
2. How to associate the address of a machine code with a line of source code.

A symbol table with this extra debugging information is called an augmented or enhanced symbol table. Because gcc and GDB run on so many different platforms, there are many different formats for debugging information:

- **stabs:** The format used by DBX on most BSD systems.
- **coff:** The format used by SDB on most System V systems before System V Release 4.
- **xcoff:** The format used by DBX on IBM RS/6000 systems.

- **dwarf:** The format used by SDB on most System V Release 4 systems.
- **dwarf2:** The format used by DBX on IRIX 6.
- **vms:** The format used by DEBUG on VMS systems.

In addition to debugging formats, GDB understands enhanced variants of these formats that allow it to make use of GNU extensions. Debugging an executable with a GNU enhanced debugging format with something other than GDB will can result in anything from it working correctly to the debugger crashing.

Don't let all these formats scare you: in the next section, I'll show you that GDB automagically picks whatever format is best for you. And for the .1% of you that need a different format, you're already knowledgeable enough to make that decision.

Preparing An Executable For Debugging

If you plan on debugging an executable, a corefile resulting from an executable, or a running process, you **must** compile the executable with an enhanced symbol table. To generate an enhanced symbol table for an executable, we must compile it with gcc's `-g` option:

```
gcc -g -o filename filename.c
```

As previously discussed, there are many different debugging formats. The actual meaning of `-g` is to produce debugging information in the native format for your system.

As an alternative to `-g`, you can also use gcc's `-ggdb` option:

```
gcc -ggdb -o filename filename.c
```

which produces debugging information in the most expressive format available, including the GNU enhanced variants previously discussed. I believe this is probably the option you want to use in most cases.

You can also give a numerical argument to `-g`, `-ggdb` and all the other debugging format options, with 1 being the least amount of information and 3 being the most. Without a numerical argument, the debug level defaults to 2. By using `-g3` you can even access preprocessor macros, which is really nice. I suggest you always use `-ggdb3` to produce an enhanced symbol table.

Debugging information compiled into an executable will not be read into memory unless GDB loads the executable. This means that executables with debug information will not run any slower than executables without debug information (a common misconception). While it's true that debugging executables take up more disk space, the executable will not have a larger "memory footprint" unless it's from within GDB. Similarly, executable load time will be nearly the same, again, unless you run the debug executable from within GDB.

One last comment. It's certainly possible to perform compiler optimizations on an executable which has an augmented symbol table, in other words: `gcc -g -O9 try1.c`. In fact, GDB is one of the few symbolic debuggers which will generally do quite well debugging optimized executables. However, you should generally turn off optimizations when debugging an executable because there are situations that will confuse GDB. Variables may get optimized out of existence, functions may get inlined, and more things may happen that may or may not confuse gdb. To be on the safe side, turn off optimization when you're debugging a program.

Exercises

1. Run " `strip --only-keep-debug try1`". Look at the file size of **try1**. Now run " `strip --strip-debug try1` and look at the file size. Now run `strip --strip-all try1` and look at the file size. Can you guess what's happening? If not, your punishment is to read "man strip", which makes for some provocative reading.
2. You stripped all the unnecessary symbols from **try1** in the previous exercise. Re-run the program to make sure it works. Now run " `strip --remove-section=.text try1`" and look at the file length. Now try to run **try1**. What do you suppose is going on?
3. Read this link about symbol tables (it's short).
4. Optional: Read this link about the COFF object file format.

Investigating The Stack With GDB

We'll look at the stack again, this time, using GDB. You may not understand all of this since you don't know about breakpoints yet, but it should be intuitive. Compile and run try1.c:

```
#include<stdio.h>
static void display(int i, int *ptr);

int main(void) {
   int x = 5;
   int *xptr = &x;
   printf("In main():\n");
   printf("   x is %d and is stored at %p.\n", x, &x);
   printf("   xptr points to %p which holds %d.\n", xptr, *xptr);
   display(x, xptr);
   return 0;
}

 void display(int z, int *zptr) {
        printf("In display():\n");
    printf("   z is %d and is stored at %p.\n", z, &z);
    printf("   zptr points to %p which holds %d.\n", zptr, *zptr);
18   }
```

Make sure you understand the output before continuing with this tutorial. Here's what I see:

```
$ ./try1
In main():
   x is 5 and is stored at 0xbffff948.
   xptr points to 0xbffff948 which holds 5.
In display():
   z is 5 and is stored at 0xbffff924.
   zptr points to 0xbffff948 which holds 5.
```

You debug an executable by invoking GDB with the name of the executable. Start a debugging session with **`try1`**. You'll see a rather verbose copyright notice:

```
$ gdb try1
GNU gdb 6.1-debian
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.

(gdb)
```

The **`(gdb)`** is GDB's prompt. It's now waiting for us to input commands. The program is currently not running; to run it, type **`run`**. This runs the program from inside GDB:

```
(gdb) run
Starting program: try1
In main():
   x is 5 and is stored at 0xbffffb34.
   xptr points to 0xbffffb34 which holds 5.
In display():
   z is 5 and is stored at 0xbffffb10.
   zptr points to 0xbffffb34 which holds 5.

Program exited normally.
(gdb)
```

Well, the program ran. It was a good start, but frankly, a little lackluster. We could've done the same thing by running the program ourself. But one thing we can't do on our own is to pause the program in the middle of execution and take a look at the stack. We'll do this next.

You get GDB to pause execution by using breakpoints. We'll cover breakpoints later, but for now, all you need to know is that when you tell GDB **`break 5`**, the program will pause at line **`5`**. You may ask: does the program execute line 5 (pause between 5 and 6) or does the program not execute line 5 (pause between 4 and 5)? The answer is that line 5 is not executed. Remember these principles:

1. **`break 5`** means to pause at line 5.
2. This means GDB pauses between lines 4 and 5. Line 4 has executed. Line 5 has not.

Set a breakpoint at line 10 and rerun the program:

```
(gdb) break 10
Breakpoint 1 at 0x8048445: file try1.c, line 10.
(gdb) run
Starting program: try1
In main():
   x is 5 and is stored at 0xbffffb34.
   xptr holds 0xbffffb34 and points to 5.

Breakpoint 1, main () at try1.c:10
10        display(x, xptr);
```

We set a breakpoint at line 10 of file try1.c. GDB told us this line of code corresponds to memory address **`0x8048445`**. We reran the program and got the first 2 lines of output. We're in **`main()`**, sitting before line 10. We can look at the stack by using GDB's **`backtrace`** command:

```
(gdb) backtrace
#0  main () at try1.c:10
(gdb)
```

There's one frame on the stack, numbered 0, and it belongs to **`main()`**. If we execute the next line of code, we'll be in **`display()`**. From the previous section, you should know exactly what should happen to the stack: another frame will be added to the bottom of the stack. Let's see this in action. You can execute the next line of code using GDB's **`step`** command:

```
(gdb) step
display (z=5, zptr=0xbffffb34) at try1.c:15
15              printf("In display():\n");
(gdb)
```

Look at the stack again, and make sure you understand everything you see:

```
(gdb) backtrace
#0  display (z=5, zptr=0xbffffb34) at try1.c:15
#1  0x08048455 in main () at try1.c:10
```

Some points to note:

- We now have two stack frames, frame 1 belonging to **`main()`** and frame 0 belong to **`display()`**.
- Each frame listing gives the arguments to that function. We see that **`main()`** took no arguments, but **`display()`** did (and we're shown the value of the arguments).
- Each frame listing gives the line number that's currently being executed within that frame. Look back at the source code and verify you understand the line numbers shown in the backtrace.
- Personally, I find the numbering system for the frame to be confusing. I'd prefer for **`main()`** to remain frame 0, and for additional frames to get higher numbers. But this is consistent with the idea that the stack grows "downward". Just remember that the lowest numbered frame is the one belonging to the most recently called function.

Execute the next two lines of code:

```
(gdb) step
In display():
16        printf("   z is %d and is stored at %p.\n", z, &z);
(gdb) step
   z is 5 and is stored at 0xbffffb10.
17        printf("   zptr holds %p and points to %d.\n", zptr, *zptr);
```

Recall that the frame is where automatic variables for the function are stored. Unless you tell it otherwise, GDB is always in the context of the frame corresponding to the currently executing function. Since execution is currently in **`display()`**, GDB is in the context of frame 0. We can ask GDB to tell us which frame its context is in by giving the **`frame`** command without arguments:

```
(gdb) frame
#0  display (z=5, zptr=0xbffffb34) at try1.c:17
17        printf("   zptr holds %p and points to %d.\n", zptr, *zptr);
```

I didn't tell you what the word "context" means; now I'll explain. Since GDB's context is in frame 0, we have access to all the local variables in frame 0. Conversely, we don't have access to automatic variables in any other frame. Let's investigate this. GDB's **`print`** command can be used to give us the value of any variable within the current frame. Since **`z`** and **`zptr`** are variables in **`display()`**, and GDB is currently in the frame for **`display()`**, we should be able to print their values:

```
(gdb) print z
$1 = 5
(gdb) print zptr
$2 = (int *) 0xbffffb34
```

But we do not have access to automatic variables stored in other frames. Try to look at the variables in **`main()`**, which is frame 1:

```
(gdb) print x
No symbol "x" in current context.
(gdb) print xptr
No symbol "xptr" in current context.
```

Now for magic. We can tell GDB to switch from frame 0 to frame 1 using the **`frame`** command with the frame number as an argument. This gives us access to the variables in frame 1. As you can guess, after switching frames, we won't have access to variables stored in frame 0. Follow along:

```
(gdb) frame 1                          <--- switch to frame 1
#1  0x08048455 in main () at try1.c:10
10        display(x, xptr);
(gdb) print x
$5 = 5                                 <--- we have access to variables in frame 1
(gdb) print xptr
$6 = (int *) 0xbffffb34                <--- we have access to variables in frame 1
(gdb) print z
No symbol "z" in current context.      <--- we don't have access to variables in frame 0
(gdb) print zptr
No symbol "zptr" in current context.   <--- we don't have access to variables in frame 0
```

By the way, one of the hardest things to get used to with GDB is seeing the program's output:

```
x is 5 and is stored at 0xbffffb34.
xptr holds 0xbffffb34 and points to 5.
```

intermixed with GDB's output:

```
Starting program: try1
In main():
...
   Breakpoint 1, main () at try1.c:10
10        display(x, xptr);
```

intermixed with your input to GDB:

```
(gdb) run
```

intermixed with your input to the program (which would've been present had we called some kind of input function). This can get confusing, but the more you use GDB, the more you get used to it. Things get tricky when the program does terminal handling (e.g. ncurses or svga libraries), but there are always ways around it.

Exercises

1. Continuing from the previous example, switch back to `display()`'s frame. Verify that you have access to automatic variables in `display()`'s frame, but not `main()`'s frame.
2. Figure out how to quit GDB on your own. Control-d works, but I want you to guess the command that quits GDB.
3. GDB has a help feature. If you type `help foo`, GDB will print a description of command foo. Enter GDB (don't give GDB any arguments) and read the help blurb for all GDB commands we've used in this section.
4. Debug try1 again and set a breakpoint anywhere in `display()`, then run the program. Figure out how to display the stack along with the values of every local variable for each frame at the same time. Hint: If you did the previous exercise, and read each blurb, this should be easy.