Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue & Thu 12:00-13:15 @ ASRC 230, Lab: Fri 10:00-11:15/11:30-12:45 @ FAB 170

Lab 7: Runtime Storage

(Adapted from Prof. Andrew Tolmach's earlier version)

In this lab, we will use `gcc` and `gdb` to explore the details of memory layout and call stack frame construction and deconstruction on the X86-64.

Download and unzip the file `lab7.zip`. It includes these notes and also a copy of the file `gdb.pdf`, about debugging assembly code with `gdb`.

Finding Memory Locations for Program Objects

The following program has been shown in this week's lecture.

```
char big_array[1<<24];  /* 16 MB */
char huge_array[1<<28]; /* 256 MB */

int beyond;
char *p1, *p2, *p3, *p4;

int useless() { return 0; }

int main()
{
  p1 = malloc(1<<28); /* 256 MB */
  p2 = malloc(1<< 8); /* 256 B */
  p3 = malloc(1<<28); /* 256 MB */
  p4 = malloc(1<< 8); /* 256 B */
  /* Some print statements ... */
}
```

Try to add `printf` statements to the program, so that the memory addresses for all variables and all allocated objects can be displayed when the program runs. Compile the program to both IA32 and X86-64 executables:

```
linux> gcc -m32 -o example0_32 example0.c
linux> gcc -o example0 example0.c
```

Run the two executable programs, `example0_32`, and `example0`, to see the print outs.

Working with gdb

Type

```
linux> make example1
```

to compile the program `example1.c`. We first compile from `example1.c` to `example1.s`, and then assemble (with `-g` set for debugging information) and link `example1.s` to the executable `example1`. By default, `gcc` uses no optimization (equivalent to `-O0`); we also compile a version with moderate optimization (`-O1`) enabled:

```
linux> make example1_O1
```

Consider `example1.c`, `example1.s`, and `example1`.

We can look at the `.s` file to see the generated code, but instead let's focus on inspecting the runtime state directly using `gdb`. See the `gdb.pdf` document for help.

To get started, type

```
linux> gdb example1
print main   [to get address of main]
break *[address of main]
run
```

This should stop at the breakpoint at the first location of `main`. (Just putting a breakpoint on symbol `main` doesn't quite work.)

Try using the `list` and `disass` commands to examine the assembly code. Note that they give similar, but not identical, information.

- `list` displays the contents of the `.s` file. It uses the same instruction mnemonics, symbolic addresses, and decimal offsets as that file.
- `disass` generates the assembly code on the fly by disassembling the binary code representation. It sometimes uses different instruction mnemonics, shows absolute addresses as well as symbolic ones, and uses hex offsets.

Use `disass` to determine the range of absolute addresses occupied by `main` (e.g. `0x400502-0x400525`).

Type

```
info registers
```

or

```
print $rsp
```

to determine the initial value of the stack pointer (e.g. `0x7fffffffe888`).

Use

```
stepi
```

to trace the execution of the program. Each step should echo the corresponding line of the `.s` file. (Note that you can execute the last command repeatedly by just typing return.)

At *each* step, determine how the stack changes. Trace how the prologue and epilogue code builds or destroy parts of the frame.

As a byproduct of this exercise, produce a memory map showing:

- Location of code for each function.
- Location of each global.
- Location of each stack frame.
- Detailed layout of each stack frame.

Some things to notice along the way:

- The generated code always uses a frame pointer, even when (as for `'g'`) it is completely unnecessary because the frame has size 0.
- As part of the prologue, the arguments, which were passed in registers, are stored into the frame, and subsequently always read/written from/to their frame locations. This rather odd behavior is because because `gcc -O0` keeps the primary copy of variables in memory rather than registers, in order to aid source-level debugging.

- Notice that the offsets of variables in the frame can be surprising, because gcc often introduces padding between them. Sometimes this padding is needed to maintain alignment of items or of the stack as a whole. According to the ABI, each value needs to be aligned to its size (e.g. 8-byte values must live on 8-byte boundaries), and `(%rsp+8)` must be 16-byte aligned at every function entry point. But sometimes the purpose of the padding is frankly mysterious.
- Use `disass` to observe how the same global variable (e.g. `'b'`) is accessed at different offsets from `%eip`, while its global address is fixed.
- Notice the interesting code sequence for the multiplication after the return into `f`.

Now try providing `example1` with some command line arguments, e.g.

```
./example1 forty plus two is forty-two
```

Although the program does not use these arguments, they are still accessible via the `argc` and `argv` parameters: `argc` gives the number of arguments and `argv` points to a null-terminated array whose elements are pointers to the argument strings. The command name counts as `argv[0]`.

Use `gdb` to discover exactly where and how the command line arguments are stored in memory. Add this information to your memory map.

A `gdb` *hint:* You give command line arguments as part of the `run` directive.

Exercise 1

Now consider `example1_O1.s` and `example1_O1`.

Look at the generated code. Notice that there are no frames or frame pointers at all; the stack is used only for return addresses.

Optionally, use `gdb` to trace through the execution of the code.

Exercise 2

Do a similar study of program `example2`.

As a byproduct, produce a memory map showing:

- Location of code for each function.
- Location of each array.
- Location of each stack frame.
- Detailed layout of each stack frame.

Something seems illegal about how the stack pointer is handled in `g`. What is it?

Now examine the code produced for `example2_O1`. How are stack frames and frame pointers used, if at all?