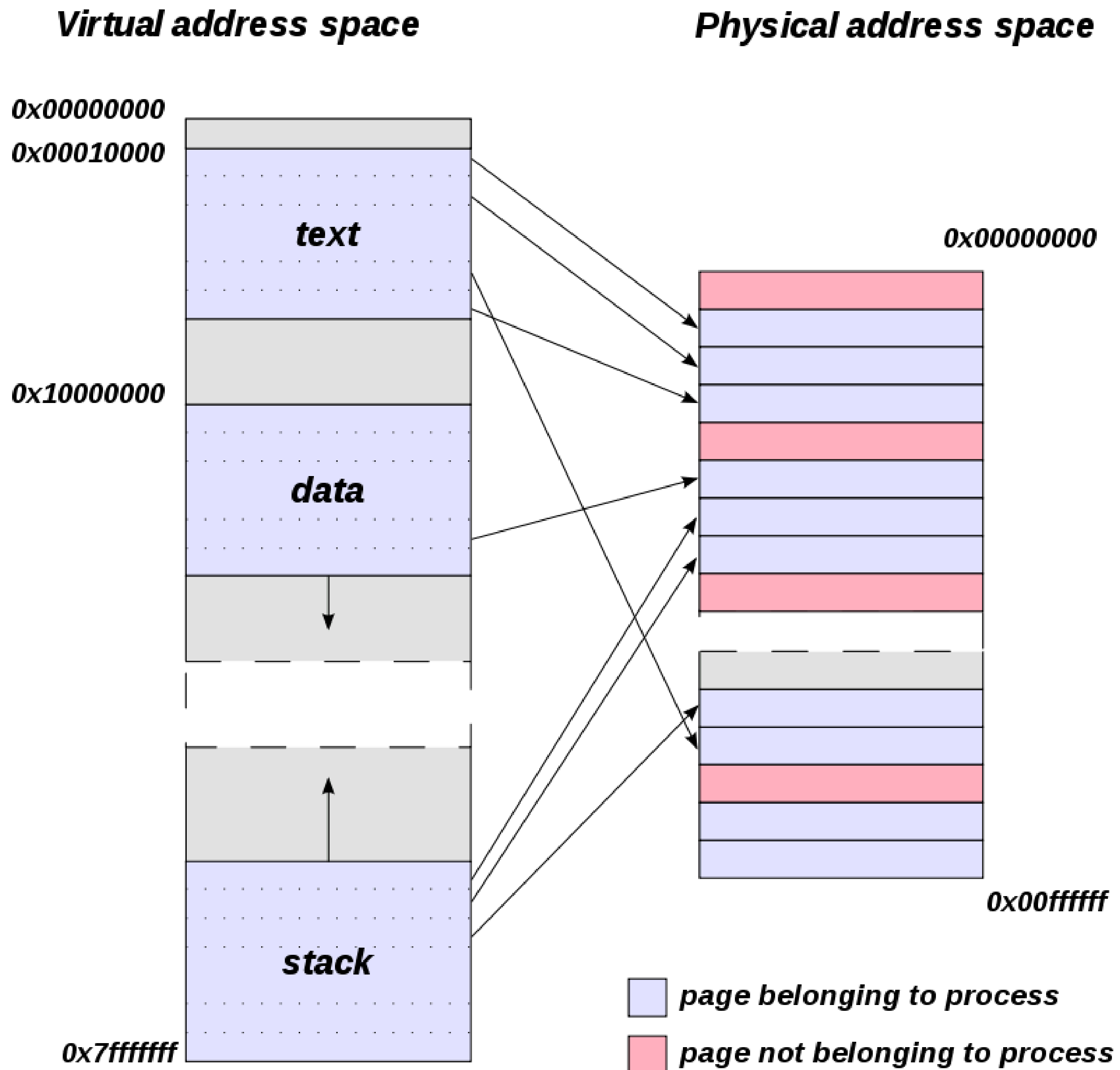
What is memory? Part 3: Registers, stacks, and threads

bytelab.codes/what-is-memory-part-3-registers-stacks-and-threads

David Vernet September 24, 2022

In the last post, we learned that a process is composed of a series of *segments*, which are loaded into its address space by a dynamic linker/loader. Once a process has been loaded and is ready to run, its address space looks something like this:



Traced by User:Stannered, original by en:User:Dysprosia, BSD <http://opensource.org/licenses/bsd-license.php>, via Wikimedia Commons

If you'll recall from the above diagram, I didn't actually explain what that "stack" box is. This is an extremely important point to understand, so we'll clarify that in this post. There's actually a lot that goes into understanding what a stack is, however. We'll have to learn about *threads*, *registers*, and even some assembly. As always though, we'll do it slowly, and we'll go over all the pertinent details. If anything in this article is confusing, just slow down, re-read whatever didn't make sense, and give it some time to sink in. If anything is confusing, I'm also just a comment away!

With that, let's learn about *threads*.

A thread is an instance of program execution on a CPU

In order for a program to run, it requires two ingredients:

1. An address space
2. Something to execute the program

We're already pretty familiar with address spaces, but what do we mean by "something to execute the program"? In software terms, we call that "something" a *thread*. A thread is conceptually just an instance of execution in a program – it is the object that runs the program's code. Recall our `lucky.c` example program from the last post:

```
// lucky_number.c
#include <stdio.h>

static int lucky_number = 7;

int main() {
    printf("My lucky number is %d!\n", lucky_number);
    lucky_number = 2;
    printf("My lucky number is now %d!\n", lucky_number);

    return 0;
}
```

For this basic program, there is a single thread which executes `main()`. This program only runs on a single CPU, and its progress is always clear to follow, and linear as the code progresses. These types of programs are called *single threaded programs*, as they're only ever run by a single execution context. There are also programs that run with multiple threads, in which case that program may run on more than one CPU at any given time. We call these programs multi-threaded, or *concurrent*. For the purposes of this discussion, we'll only focus on single threaded programs.

Alright, so we have a mental model of a thread as a "thing that executes a program's code on a CPU." That level of detail is useful, but it's not enough. Let's pull back the veil a bit, and learn what it really means for a thread to run on a CPU. In reality, a thread is actually a collection of what are called *registers*, and those registers are how software really controls a CPU.

A register is a little piece of storage on a central processing unit (CPU)

One thing you quickly realize when learning systems engineering is that a computer is a complex, cooperative dance between hardware and software. Hardware provides an interface for itself to be programmed and configured, and software responds accordingly by writing bytes to the hardware in order to achieve some desired effect.

Registers are one such interface. In similar fashion to an address space, a register is just a piece of storage. Unlike an address space, however, registers are located **on** the actual CPU itself, rather than in "main memory", or DRAM. Software can write to the registers using *instructions*, which as we learned in the last post, are actually just bytes in a process's `.text` segment. Let’s take another look at the assembly output of our test program, `lucky_number` to refresh our memory:

```
$ objdump -d lucky_number
lucky_number:     file format elf64-x86-64


…

Disassembly of section .text:
0000000000401040 <_start>:
401040:       f3 0f 1e fa             endbr64
401044:       31 ed                   xor    %ebp,%ebp
401046:       49 89 d1                mov    %rdx,%r9
401049:       5e                      pop    %rsi
40104a:       48 89 e2                mov    %rsp,%rdx
40104d:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
401051:       50                      push   %rax

…

0000000000401130 main:
401130: 55                            pushq   %rbp
401131: 48 89 e5                      movq    %rsp, %rbp
401134: 48 83 ec 10                   subq    $16, %rsp
401138: c7 45 fc 00 00 00 00          movl    $0, -4(%rbp)
40113f: 8b 34 25 30 40 40 00          movl    4210736, %esi
```

```
401146: 48 bf 04 20 40 00 00 00 00 00 movabsq $4202500, %rdi
401150: b0 00                         movb    $0, %al
401152: e8 d9 fe ff ff                callq   -295 <printf@plt>
401157: c7 04 25 30 40 40 00 02 00 00 00       movl    $2, 4210736
401162: 8b 34 25 30 40 40 00          movl    4210736, %esi
401169: 48 bf 1c 20 40 00 00 00 00 00 movabsq $4202524, %rdi
401173: 89 45 f8                      movl    %eax, -8(%rbp)
401176: b0 00                         movb    $0, %al
401178: e8 b3 fe ff ff                callq   -333 <printf@plt>
40117d: 31 c9                         xorl    %ecx, %ecx
40117f: 89 45 f4                      movl    %eax, -12(%rbp)
401182: 89 c8                         movl    %ecx, %eax
401184: 48 83 c4 10                   addq    $16, %rsp
401188: 5d                            popq    %rbp
401189: c3                            retq
40118a: 66 0f 1f 44 00 00             nopw    (%rax,%rax)
```

Recall that those numbers in the middle column collectively represent the program's *instructions*, in pure numerical form. The rightmost two columns show those same instructions, but represented in human readable assembly language. In this case, we're looking at the instructions that comprise the `_start()` and `main()` functions of our program.

These instructions happen to be [x86-64 instruction set architecture (ISA)](), but different types of computers (i.e. different *architectures*) have different instruction sets. x86-64 is known as a [Complex Instruction Set Computer (CISC)]() architecture, so it has many instructions which can do all sorts of fancy and complicated things. For our purposes, the important thing to know about x86-64 instructions is that they can operate on **registers**, and **memory**.

Speaking of registers – if you look at the rightmost column, notice all of those three-letter jumbled words which are preceded by a %, such as `%rsp`? Those are registers, and those three letters are how we identify which register an instruction is reading or writing. For example, `%rbp` is called the "base pointer register", and `%rsp` is the "stack pointer register". The second column from the right, which has a bunch of short mnemonic looking words, contains the name of the instructions.

Assembly language can be pretty complicated and difficult for a human to follow, so the best thing for us to do is to just jump into it and start learning. Let's take a look at three of the instructions in the `main()` function:

```
…
401131: 48 89 e5                      movq    %rsp, %rbp
401134: 48 83 ec 10                   subq    $16, %rsp
401138: c7 45 fc 00 00 00 00          movl    $0, -4(%rbp)
…
```

Let's go over the instructions one by one.

```
401131: 48 89 e5                    movq    %rsp, %rbp
```

This first instruction writes the 8-byte value currently stored in the `%rsp` register, into the `%rbp` register. It "moves" the value in `%rsp` into `%rbp`. Whatever was previously in `%rbp` is now overwritten, and replaced with the value stored in `%rsp`. We'll go into more detail about this later, but what this is really doing is recording the thread's "stack pointer", i.e. a pointer into that stack box from the picture at the beginning of the article, into `%rbp` so we can remember it for later. It's not important to understand this for now – we'll discuss it in more detail later.

```
401134: 48 83 ec 10                 subq    $16, %rsp
```

This next instruction subtracts the *immediate* (i.e. numerical) value 16 from the 8-byte value that's currently stored in the register named `%rsp`, and then writes the result to `%rsp`. After this instruction, the value stored in the `%rsp` register has been decremented by 16. Worry not though, because the original value of `%rsp` is still stored in `%rbp` thanks to the first instruction we covered above. If it helps, this instruction is conceptually the exact same thing as `%rsp -= 16`.

```
401138: c7 45 fc 00 00 00 00        movl    $0, -4(%rbp)
```

The third instruction is a bit more complicated, so let's take it step by step. First, note that the register `%rbp` stores a memory address – specifically, the address that was previously stored in `%rsp` at the first instruction above. In other words, the `%rbp` register, which is an 8-byte piece of storage that lives on the CPU, is itself storing a memory address in the process's address space – the thread's previous "stack" address, specifically.

Hopefully that makes sense. If it doesn't, go through the above instructions again. The point is that `%rbp` is storing the address that was previously stored in `%rsp`, so any operations we do to `%rbp`, are being done on that same address. Moving on, `movl    $0` is saying, "Store the value 0". The rest of the instruction is telling us **where** to store that 0. `(%rbp)` tells us to write 0 to the address stored in the register `%rbp`, and the `-4` tells us to subtract 4 from the address stored in `%rbp` to determine the final address. In other words, the whole instruction is, "*Store the value 0 at the memory address stored in `%rbp`, but subtract 4 from that memory address to get the final address where 0 should be stored*". Finally, the `l` in `movl` specifies the *size* of the instruction. In this case, it's telling us to store the 0 in only 4 bytes. Unlike the `subq` above, which ends with `q`, which specifies that we should write 8 bytes.

A very reasonable question to ask would be *why is the program doing all this*? We'll answer that more substantively later in the article. For now, we've already covered a lot, so let's take a moment to recap the most important points:

1. Registers are little pieces of storage that live on the CPU.
2. Registers can be read and written using *instructions*.
3. Memory can also be read and written, often via registers, using *instructions*.

Congratulations, you've officially read and understood x86-64 assembly language. If it doesn't all make sense yet, don't stress about it too much. Try re-reading the above section, and if it still doesn't make sense, just keep reading through the article. It's fine to let your knowledge stay a bit superficial here, as the important thing is understanding the concepts. There are also a lot of details that we haven't covered yet which we'll get to throughout the rest of the article, and in future posts.

A thread is really just a set of registers

As we saw from looking at our example program above, when a CPU is executing code, all that it's really doing is moving bytes around between registers and memory. One instruction may subtract some value from a register, and then another may write a value to the new memory address stored in that register. This, in effect, encapsulates what "executing on a CPU" means. Now that we understand that, the hard part is over. We just need to fill in a few blanks.

Let’s start filling in those blanks by talking about another register: `%rip`. We only talked about two registers in our example above: `%rsp` and `%rbp`. In x86-64, there are actually many more registers. We won’t cover all of them in this article, but there is one specifically that we need to know about now, and that's `%rip`, or the *instruction pointer register*. This is a special register which contains the address of the instruction that the CPU is currently executing. Going back to our example above, if `%rip` contains the value `0x401130`, then it is pointing at the first instruction of `main()`, and the next thing it executes will be `pushq    %rbp`. After it does that, the CPU will automatically update `%rip` to point to the next instruction at `0x401131`, which is the `movq     %rsp, %rbp` we analyzed together.

This goes back to the point above: "a computer is a complex, cooperative dance between hardware and software." In this case, software has loaded some instructions into memory, and then pointed the CPU at those instructions by loading a value into `%rip`. With that, we can now formally define "thread" in a more concrete way: *a thread is an instance of a set of registers in a program*. For our `lucky_number.c` program, a thread is a set of `%rip`, `%rsp`, `%rbp`, …, registers which will be used to program the hardware to actually execute the program. When the `lucky_number.c` program is loaded and is ready to run, the kernel will allocate a set of registers for the program’s initial thread, load its start address into `%rip`, and then that thread will start executing on the CPU. For the case of `lucky_number.c`, the kernel will load `0x401040`, the address of `_start()`, into the initial thread’s `%rip`.

Every thread has a range of memory called a "stack"

We’ve made a lot of progress so far. We now understand that a thread is a set of registers that allows it to function as an instance of execution on a CPU for a program. For the case of `lucky_number.c`, this thread will issue a couple of `printf()` calls, and write the value `2` to the global `lucky_number` variable.

But let’s take a step back for a moment. We learned in our last posts that a dynamic linker/loader will store global variables in various segments of a process's address space. For example, the `lucky_number` variable in our example above is stored in `.data`. But where do local variables go? For example, a program can declare a variable inside of a loop, or a function may declare a local variable that can only be referenced inside of that function. In both cases, the variables are temporary, and do not have global visibility. So where in the process's address space are those variables stored? The answer is that they're stored on the executing thread's *stack*. Let’s take a look a new example, `local_scope.c`, to see what we mean:

```
// local_scope.c
#include <stdio.h>

static int add_eight(int num) {
    int sum;
    sum = num + 8;

    return sum;
}

int main() {
    int my_num = 8;

    return add_eight(8);
}
```

In `add_eight()`, the variable `sum` is local, and in `main()`, `my_num` is local. Neither of these variables have global scope, so they'll both be stored on the executing thread's *stack*. But what exactly is this "stack"?

A thread's *stack* is a range of memory that the operating system allocates for it when the thread is created, and which is used to store local state while the thread is executing the program. When it first begins executing, a thread starts at the top of its stack by pointing its `%rsp` to the top-most address of that range of memory, and then moves downward as execution progresses. Let's take a look at the disassembly of `local_scope.c` to put this in context:

```
local_scope:     file format elf64-x86-64
Disassembly of section .text:

...

0000000000001119 <add_eight>:
1119:       55                      push   %rbp
111a:       48 89 e5                mov    %rsp,%rbp
111d:       48 83 ec 18             sub    $0x18,%rsp
1121:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
1125:       48 8b 45 e8             mov    -0x18(%rbp),%rax
1129:       48 83 c0 08             add    $0x8,%rax
112d:       48 89 45 f8             mov    %rax,-0x8(%rbp)
1131:       48 8b 45 f8             mov    -0x8(%rbp),%rax
1135:       c9                      leave
1136:       c3                      ret

0000000000001133 <main>:
1133:       55                      push   %rbp
1134:       48 89 e5                mov    %rsp,%rbp
1137:       48 83 ec 10             sub    $0x10,%rsp
113b:       48 c7 45 f8 08 00 00    movq   $0x8,-0x8(%rbp)
1142:       00
1143:       48 8b 45 f8             mov    -0x8(%rbp),%rax
1147:       48 89 c7                mov    %rax,%rdi
114a:       e8 ca ff ff ff          call   1119 <add_eight>
114f:       c9                      leave
1150:       c3                      ret
```

We once again have some complex looking assembly code. Let's start by looking at the instructions at `main()`.

```
1133:       55                      push   %rbp
```

This first instruction, `push`, is a special instruction that operates on the stack pointer `%rsp`. `push` does two things:

1. Decrement the value of `%rsp` by 8 bytes. That is, it updates `%rsp` to contain an address that is 8 bytes lower than whatever was stored there previously.
2. It stores the 8-byte value of `%rbp` at the memory address pointed to by the newly decremented `%rsp`.

So if `%rsp` had the value `0x20`, then push `%rbp` would do two things:

1. Decrement `%rsp` to `0x18`.
2. Store the 8-byte value in `%rbp`, at the memory address `0x18`.

In plain-English, this has the effect of storing the value of `%rbp` on the stack so we can reference it later. We'll see below that this is how the program stores its *frame pointer*. Before discussing that in more detail though, let’s move onto the next instruction.

```
1134:       48 89 e5                mov    %rsp,%rbp
```

We've seen the `mov` instruction before, though thankfully this example is simpler than the one from above. This moves the current value of `%rsp` (which was decremented in the `push` instruction above) into `%rbp`, overwriting whatever was there before. This isn't a problem though, because the previous value of `%rbp` is safely stored on the stack thanks to the `push` instruction!

```
1137:       48 83 ec 10             sub    $0x10,%rsp
```

We're now subtracting an additional 16 bytes from `%rsp`. After this instruction, `%rsp` has been moved "down" a total of 24 bytes since the function started. With this latest instruction, we now have 16 bytes of unused space on the stack that we could store things in.

```
113b:       48 c7 45 f8 08 00 00    movq   $0x8,-0x8(%rbp)
```

Alright, this one is a bit tricky. We should recognize this instruction, as we've seen it before. `movq    $0x8,-0x8(%rbp)`, in English terms, is storing the number 8 at the address stored in `%rbp`, adjusted down by 8. Why would the compiler do this? Well, recall that above, we stored the value of `%rsp` in `%rbp`, and then we decremented `%rsp` by 16. So `%rbp == (%rsp + 16)`. This means that `%rbp - 8 == %rsp + 8`, so if we’re storing an 8 byte number at `(%rbp - 8)`, we’re actually storing a number in one of the two 8 byte slots that are now available to us on the stack, thanks to having decremented `%rsp` in the prior instruction.

And why are we storing the number 8 on the stack? Because it's the local variable `int my_num` from our `local_scope.c` program. Pretty neat, right? Let’s move on.

```
1142:       00
1143:       48 8b 45 f8             mov    -0x8(%rbp),%rax
```

This instruction should look somewhat familiar as well. It's another `mov` instruction, but this time, we're loading a value from memory and storing it in a register, rather than writing a value to a memory address that's stored in a register. In English terms, this instruction is taking the value contained in memory at the address of `(%rbp - 8)`, and storing it in the `%rax` register. We know what’s contained in `%rbp - 8`, because we just stored the number 8 there in the last instruction. So this instruction is taking the value 8 we stored in the last instruction, and writing it to `%rax`. After this instruction, `%rax == 8`.

```
1147:       48 89 c7                mov    %rax,%rdi
```

This instruction is another straightforward one. We're copying the value stored in `%rax`, and putting it in `%rdi`. After this instruction, `%rdi == 8`, because `%rax == 8` thanks to the instruction above.

Now, why would we do this? We won't go into too much detail on what's going on here in this post, as it's a bit of a rabbit hole, but the short answer is that the compiler is applying what we call an *x86 calling convention*. In the x86-64 ISA, when a function is called, it expects that the `%rdi` register will contain its first argument. If we peek at the next instruction, we see that we're issuing a `call` instruction to the `add_eight()` function. So in effect, this instruction is storing the value in `%rdi` so that the `add_eight()` function can read it after being called. Speaking of the `call` instruction, let's move on.

```
114a:       e8 ca ff ff ff          call   1119 <add_eight>
```

As mentioned above, this is the `call` instruction. As you might have guessed, this instruction "calls" another function in the process's address space, and changes the *control flow* of the program to have `%rip` point to that function. This instruction actually performs two steps under the hod:

1. `pushq <address of instruction following call in current function>`
2. Update `%rip` to the address of the new function

In other words, it pushes the address of the instruction **following** the `call` instruction onto the stack. It then updates `%rip` to point to the address of the new function, and then starts executing.

Because `call` is updating `%rip` to point to the first instruction in `add_eight()`, let's start walking through those instructions. As a reminder, here's the disassembly of `add_eight()`:

```
0000000000001119 <add_eight>:
1119:       55                      push   %rbp
111a:       48 89 e5                mov    %rsp,%rbp
111d:       48 83 ec 18             sub    $0x18,%rsp
1121:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
1125:       48 8b 45 e8             mov    -0x18(%rbp),%rax
1129:       48 83 c0 08             add    $0x8,%rax
112d:       48 89 45 f8             mov    %rax,-0x8(%rbp)
1131:       48 8b 45 f8             mov    -0x8(%rbp),%rax
1135:       c9                      leave
1136:       c3                      ret
```

Let's begin with the first instruction:

```
1119:       55                      push   %rbp
```

The first instruction we should recognize, as it's the exact same as the first instruction in `main()`. As a reminder, this instruction decrements `%rsp` by 8, and stores the current value of `%rbp` there. But what was in `%rbp` before? If you'll recall from above, `%rbp` contained the address on the stack where we previously stored `%rbp` in `main()`. Thus, if we had to, we could access that initial value of `%rbp` in `main()` by reading `(%rbp)`, and reading the 8-byte value stored at that address. Those 8 bytes would actually refer to the **previous** `%rbp`, as we stored that value on the stack in the first `push %rbp` instruction in `main()`!

This technique can be applied recursively as well. We can walk up the stack by reading `%rbp`, and then inspecting the value on the stack that's stored there to get the next value of `%rbp`, etc. This is where `%rbp`, or "base pointer register", gets its name. The register points to the "base" of the stack for a given function invocation. This technique of using `%rbp` to store the address of the stack pointer when a function is invoked is called *frame pointers*. This is a key technique in implementing debuggers, profilers, and more, which all need to walk a thread's callstack.

Let’s move onto the next instructions.

```
111a:       48 89 e5                mov    %rsp,%rbp
```

Again, as we did in `main()`, we're storing the current value of `%rsp` into `%rbp`. Not to worry though, as mentioned above, if we need to read the old value of `%rbp`, we simply have to read the value stored at the 8 bytes of stack memory that’s now stored in `%rbp`.

```
111d:       48 83 ec 18             sub    $0x18,%rsp
```

Here, we're again subtracting some bytes from `%rsp` which we can use as available memory on the stack for local variables. This time, we're giving ourselves `0x18 == 24` bytes.

```
1121:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
```

Recall that as mentioned above, `%rdi` contains the first argument to the `add_eight()` function according to x86-64 calling conventions. This instruction therefore stores the first argument (8 in our example) as an 8-byte value onto the stack.

```
1125:       48 8b 45 e8             mov    -0x18(%rbp),%rax
```

This instruction now takes the value 8 that we just stored in memory on the stack, and puts it in the register `%rax`. What does this accomplish? Well, `%rax` is also a special register. Just like how `%rdi` stores the first argument of a function in x86-64 calling convention, `%rax` stores the return value of the function. We're going to be returning 8 + 8 == 16, so this instruction stores the first of those two eights in the equation – the one that was passed as an argument to `add_eight()`.

```
1129:       48 83 c0 08             add    $0x8,%rax
```

Here's the second of those two eights. This instruction, as you probably guessed, adds 8 to the current value of `%rax`. `%rax` had the value eight before, so now it has the value 16. That's the value that we want to return to the caller, so the function should now be able to start cleaning up and returning to `main()`.

```
112d:       48 89 45 f8             mov    %rax,-0x8(%rbp)
1131:       48 8b 45 f8             mov    -0x8(%rbp),%rax
```

These two instructions are a bit wonky. The compiler is storing the value in `%rax` (which is 16 after our last two instructions), into one of the empty slots on the stack. In the next instruction, the compiler is again copying that value directly back into `%rax`.

But...why? The reason the compiler is doing this is that I compiled the program with `-O0`, meaning without any optimization. This directs the compiler to do the absolute simplest possible thing, even if it doesn't make much sense and something obviously more performant is available. In an optimized program, the compiler would never do this, so for all practical purposes we can just ignore it.

```
1135:       c9                      leave
```

`leave` is a special x86 instruction that "unwraps" a frame pointer. When we first entered the function, we did a `push %rbp`, followed by a `mov %rbp, %rsp`. The `leave` instruction resets `%rsp` and `%rbp` to what they were at the beginning of the function. In essence, `leave` is identical to doing:

```
mov %rbp, %rsp
pop %rbp
```

`pop` is of course the complementary instruction of `push`. It reads the value currently on the stack and stores it in `%rbp`, and then increments `%rsp` by 8. Let's move onto the final instruction in `add_eight()`:

```
1136:       c3                      ret
```

The `ret` instruction is how we return to the calling function. Recall that the `call` instruction stores the address of the next instruction following the `call` to `add_eight()`, on the stack. We've reset `%rsp` to point to that exact address, so by issuing `ret`, we're instructing the CPU to read the instruction currently located on the stack, write it to `%rip`, and then increment `%rsp` by 8.

This brings us back to the end of the `main()` function. Thankfully, the last two instructions in `main()` will look familiar:

```
114f:       c9                          leave
1150:       c3                          ret
```

That's right, they're the exact same as the last two instructions in `add_eight()`. The `leave` instruction will reset `%rsp` and `%rbp` to what they were at the beginning of `main()` before we stored the frame pointer, and then `ret` will update `%rip` to point to the instruction following the `call` instruction that called `main()`. Note as well that the value 16 is still stored in `%rax`, as `add_eight()` stored 16 to `%rax`, and `main()` didn't update it before returning. If the caller inspected `%rax`, it would therefore see 16, which has the same effect as `main()` returning 16.

That's what a program really is

Ok, so we just covered a ***lot*** of low level details about how threads, function, and computers in general actually work. Let's recap some of the high level points:

1. A thread is a set of registers, along with a stack.
2. As a thread executes, it reads and writes registers and memory.
3. The thread's stack is the range of memory that it "executes on". As it calls functions, the thread walks **down** and consumes its stack. As it returns from functions, it walks back up its stack.
4. Local variables are stored on the stack. In general, the stack is available to the thread for any local, scratch space that's required by the program.
5. When a function is invoked, the thread stores a *frame pointer* which it can use to walk back up its stack whenever it needs. This frame pointer is recorded by storing the current value of `%rbp` at the top of the stack stack when the function is first invoked, and then storing a pointer back to that stack value in `%rbp`.
6. x86-64 works using a *calling convention*. Functions expect specific registers to hold their arguments (e.g. `%rdi` for the first argument), and populate the `%rax` register with the return value of the function call.

At the end of the day, that's *basically* how a program runs.

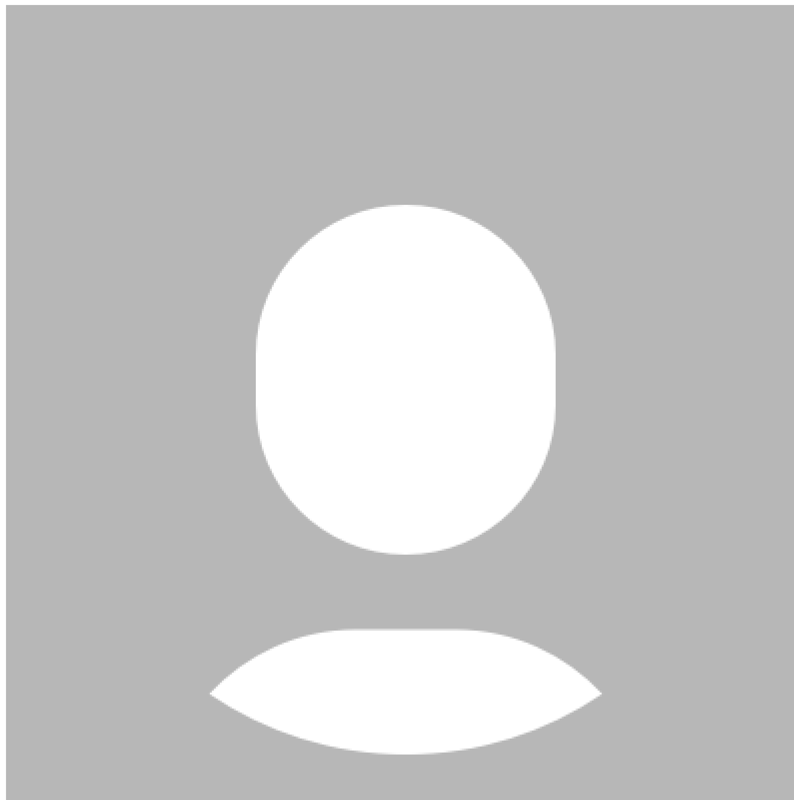
There is a lot more detail to cover, but if you've made it this far, you've honestly already done the hard part. The rest of the story is just continuing to fill in blanks. In the next article, we'll talk about some of the other registers that we glossed over, and go into more details of the full x86-64 calling convention. For example, there are of course other registers that store the other possible arguments to a function, and we'll learn about all of those. We'll also learn that registers can have different widths, just as instructions can.

There's a lot more to learn, but we're well on our way at this point to really understanding how computer systems work. I hope you enjoyed the article. Let me know in the comments if

anything was unclear, and have a great day!

1 Comments

to join the conversation.



Anmol 3 years ago

Thank you very much David for these articles. As a systems student, these articles are very useful for filling the knowledge gaps, and learn some new things.

♡ 0

Powered by Cove