Analyzing Stack Frames Using Recursion

Anthony Ramos

CSC342/43 – Spring 2021

Quick Summary:

A Stack Frame is a frame of data (i.e., a region of memory) that gets created (i.e, allocated) on the stack (think of it as a stack within a stack). Each time a function is called, a new stack frame is created. Each *new* frame initialized by pushing the old base pointer onto the stack. A new base pointer is initialized, and space is allocated (via a subtraction) from stack pointer for necessary data. Next, any arguments are pushed onto the stack (using negative offsets w.r.t base pointer for local variables). Lastly, the *return address* (i.e., the address after the function call) is pushed to the top of the stack prior moving on to the next function call. For each function call, this process repeats for a new Stack Frame. The behavior of Stack Frames may differ depending on the program. One of the best ways to understand Stack Frames is through recursion. In this exercise, we will analyze the stack frames of a C program that computes the factorial of some integer using Linux GCC/GDB on an Intel x64-Bit processor.

Additional Resources:

- Chapter 2.8 of textbook (Nested Procedures) (Pg 100 or Pg 123 for PDF)
- Overview of Stack Frames
- Another overview of stack frames
- Recursion & Stack Frames (Quick Intro)

Factorial.c

Consider the following C program used to compute the factorial of some integer. We assign fact() with argument 3 to local variable a. In addition, we do a print statement just for the purpose of testing the output to the console.

```c
#include <stdio.h>

int fact(int);

int main() {
        int a = fact(3);
        printf("Result: %d \n", a);

        return 0;
}

int fact (int n) {
        if (n < 1) return (1);
        else return (n * fact(n - 1));
}
```

Figure 1: A C program to compute a factorial

Main Call

```
Dump of assembler code for function main:
   0x000055555555464a <+0>:     push   %rbp
   0x000055555555464b <+1>:     mov    %rsp,%rbp
   0x000055555555464e <+4>:     sub    $0x10,%rsp
=> 0x0000555555554652 <+8>:     mov    $0x3,%edi
   0x0000555555554657 <+13>:    callq  0x55555555467c <fact>
   0x000055555555465c <+18>:    mov    %eax,-0x4(%rbp)
   0x000055555555465f <+21>:    mov    -0x4(%rbp),%eax
   0x0000555555554662 <+24>:    mov    %eax,%esi
   0x0000555555554664 <+26>:    lea    0xc9(%rip),%rdi        # 0x555555554734
   0x000055555555466b <+33>:    mov    $0x0,%eax
   0x0000555555554670 <+38>:    callq  0x555555554520 <printf@plt>
   0x0000555555554675 <+43>:    mov    $0x0,%eax
   0x000055555555467a <+48>:    leaveq
   0x000055555555467b <+49>:    retq
End of assembler dump.
```

Figure 2: Assembler Code for `main()`

At the start of the `main()` function, the first set of instructions (in orange) are executed. A stack frame is initialized with initial base pointer (i.e. old based pointer) value 0x7FFFFFFFE0C0 (in white). The *new* base pointer pushed onto the stack has value 0x7FFFFFFFDFE0 (in magenta). A subtraction is performed on the stack pointer to allocate space (16 bytes) for variables (i.e. local variable a). The stack pointer has value 0x7FFFFFFFDFD0 as a result (in green). The argument value 3 is also moved onto register `$edi` but *not* pushed onto the stack (unlike MS Visual Studio). No where in this program will registers be stored on the stack!

```
(gdb) x/xg ($rbp)
0x7fffffffdfe0: 0x00005555555546b0
(gdb) print /x $rbp
$2 = 0x7fffffffdfe0
(gdb) x/xg ($rsp)
0x7fffffffdfd0: 0x00007fffffffe0c0
(gdb) print /x $rsp
$3 = 0x7fffffffdfd0
```

```
(gdb) print /x $edi
$5 = 0x3
```

Figure 3: Initial stack frame and initial argument value

The next instruction (in blue) performs the execution of the `callq` instruction which pushes the *return address* 0x55555555465C (i.e., the instruction after `callq`) onto the stack. This is the last operation executed in `main()` prior to jumping to `fact()`.

```
(gdb) x/xg ($rsp)
0x7fffffffdfd8: 0x000055555555465c
```

Figure 4: Pushing the Return Address to stack

First Call

The assembler code for the `fact()` function is given below. Notice the instruction at 0x55555555469C is a call to the same function `fact()`. Hence, we know that this is a recursive function.

```
Dump of assembler code for function fact:
=> 0x000055555555467c <+0>:     push   %rbp
   0x000055555555467d <+1>:     mov    %rsp,%rbp
   0x0000555555554680 <+4>:     sub    $0x10,%rsp
   0x0000555555554684 <+8>:     mov    %edi,-0x4(%rbp)
   0x0000555555554687 <+11>:    cmpl   $0x0,-0x4(%rbp)
   0x000055555555468b <+15>:    jg     0x555555554694 <fact+24>
   0x000055555555468d <+17>:    mov    $0x1,%eax
   0x0000555555554692 <+22>:    jmp    0x55555555546a5 <fact+41>
   0x0000555555554694 <+24>:    mov    -0x4(%rbp),%eax
   0x0000555555554697 <+27>:    sub    $0x1,%eax
   0x000055555555469a <+30>:    mov    %eax,%edi
   0x000055555555469c <+32>:    callq  0x55555555467c <fact>
   0x00005555555546a1 <+37>:    imul   -0x4(%rbp),%eax
   0x00005555555546a5 <+41>:    leaveq
   0x00005555555546a6 <+42>:    retq
End of assembler dump.
```

Figure 5: Assembler Code for `fact()`

At the beginning of this function call, the instructions (in orange) create a new stack frame as shown in figure 6 below. The *now* old base pointer 0x7FFFFFFFDFE0 (in white) is pushed onto the stack. The *new* base and stack pointers have values 0x7FFFFFFFDFC0 (in magenta) and 0x7FFFFFFFDFB0 (in green). Note that our argument value 0x03 (in purple) initialized in `main()` and previously stored in register `$edi` is also pushed onto the stack.

```
(gdb) x/2xg ($rsp)
0x7fffffffdfb0: 0x00007ffff7de3b40      0x0000000300000000
(gdb) print /x $rsp
$4 = 0x7fffffffdfb0
(gdb) x/2xg ($rbp)
0x7fffffffdfc0: 0x00007fffffffdfe0      0x000055555555465c
(gdb) print /x $rbp
$5 = 0x7fffffffdfc0
```

Figure 6: Stack Frame (First Call)

The next instructions in blue will compare whether or not our argument (0x03) is greater than 1. If greater than 1 (which it is), then a jump will occur from 0x555555554684 to 0x55555555468d. After jumping to instruction at address 0x555555554694, the next instructions (in yellow) will first move the argument that was previously pushed onto the stack (i.e. 0x03) to register `$eax`,

decrement it by 1, and move the result to register `$edi`. The value now stored in this register (i.e. 0x02) will become the new argument of our `fact()` function in the next call.

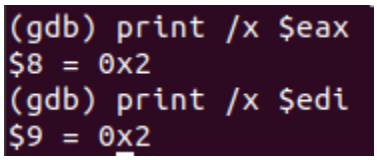

Figure 7: Register values of `$eax` and `$edi` after first call

The execution of `callq` will terminate this function call by pushing the return address 0x5555555546A1 (i.e., the instruction of `imul` after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDFC0 (which occurs in the next call)

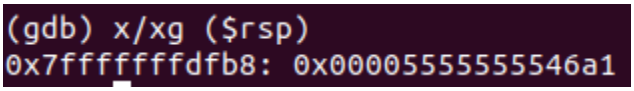

Figure 8: Pushing the return address onto the stack (after first call)

Second Call

In the next call, we remain within the function `fact()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFDFC0 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDFA0 (in magenta) and stack pointer 0x7FFFFFFDF90 (in green). Note that the updated argument value 0x02 (in purple) is also pushed onto the stack.

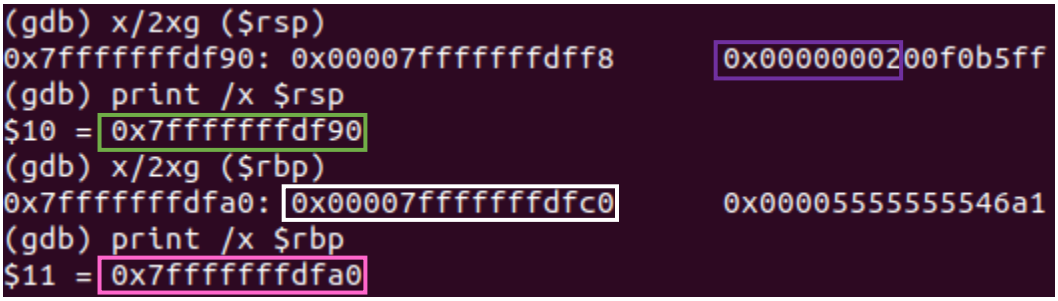

Figure 9: Stack Frame (Second Call)

The next set of instructions in blue will compare the argument value (i.e., 0x02) and perform a jump to 0x555555554694 since it is greater than 1. The next instructions in yellow will first move the argument that was previously pushed onto the stack (i.e. 0x02) to register `$eax`, decrement it by 1, and move the result to register `$edi`. The value now stored in this register (i.e., 0x01) will become the new argument of our `fact()` function in the next call.

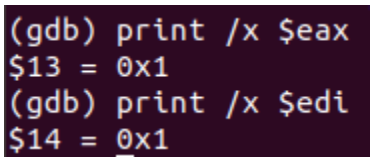

Figure 10: Register values of `$eax` and `$edi` after second call

The execution of `callq` will terminate this function call by pushing the return address 0x5555555546A1 (i.e., the instruction of `imul` after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDFA0.

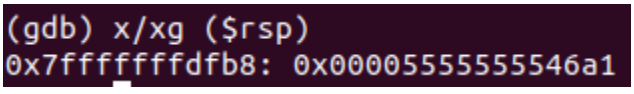

Figure 11: Pushing the return address onto the stack (after second call)

Third Call

In the next call, we remain within the function `fact()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFDFA0 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDF80 (in magenta) and stack pointer 0x7FFFFFFDF70 (in green). Note that the updated argument value 0x01 (in purple) is also pushed onto the stack.

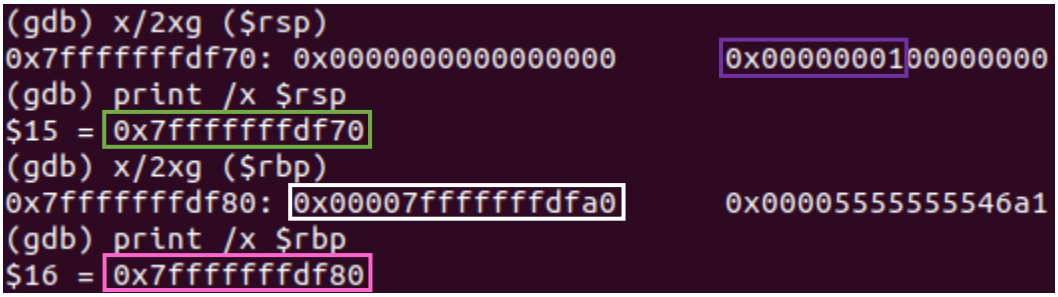

Figure 12: Stack Frame (Third Call)

The next instructions in blue will compare whether or not our argument (0x01) is greater than 1. After jumping to instruction at address 0x555555554694, the next instructions in yellow will first move the argument that was previously pushed onto the stack (i.e. x01) to register `$eax`, decrement it by 1, and move the result to register `$edi`. The value now stored in this register (i.e. 0x0) will become the new argument of our `fact()` function in the next call.

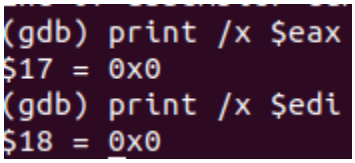

Figure 13: Register values of `$eax` and `$edi` after third call

The execution of `callq` will terminate this function call by pushing the return address 0x5555555546A1 (i.e., the instruction of `imul` after `callq`) onto the stack pointer prior to pushing the base pointer 0x7FFFFFFFDF80.

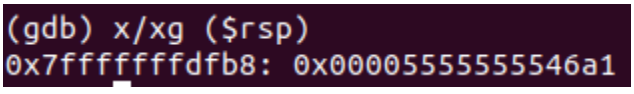

Figure 14: Pushing the return address onto the stack (after third call)

Fourth Call

In the next call, we remain within the function `fact()` and return to the set of instructions in orange. The old base pointer 0x7FFFFFFFDF80 (in white) is pushed onto the stack. A *new* stack frame is created once again with base pointer 0x7FFFFFFFDF60 (in magenta) and stack pointer 0x7FFFFFFDF50 (in green). Note that the updated argument value 0x00 (in purple) is also pushed onto the stack.

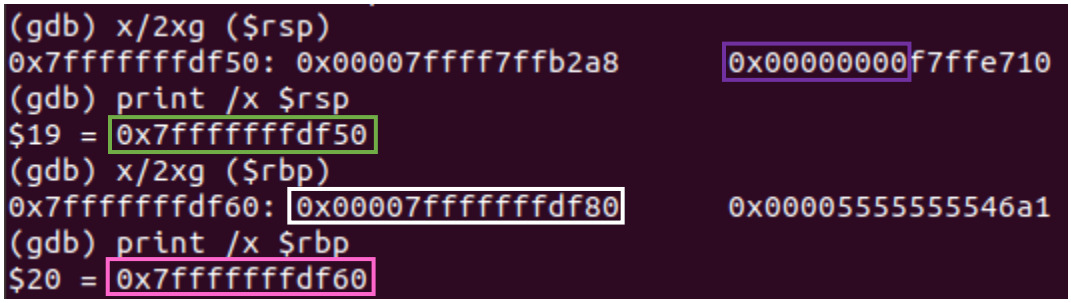

Figure 15: Stack Frame (Fourth Call)

In the next set of instructions in blue, the value of the argument is now 0x00 (i.e., less than 1) and so no jump will occur. Figure 16 below shows all stack frames that were used in the program. The stack pointers, base pointers, return addresses, and argument values are shown in green, magenta, amber, and purple respectively.

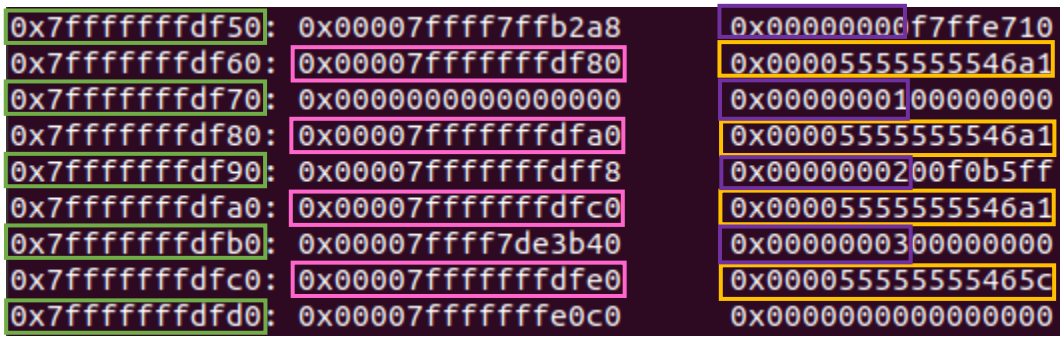

Figure 16: Stack frames of entire program

Return to Third Call

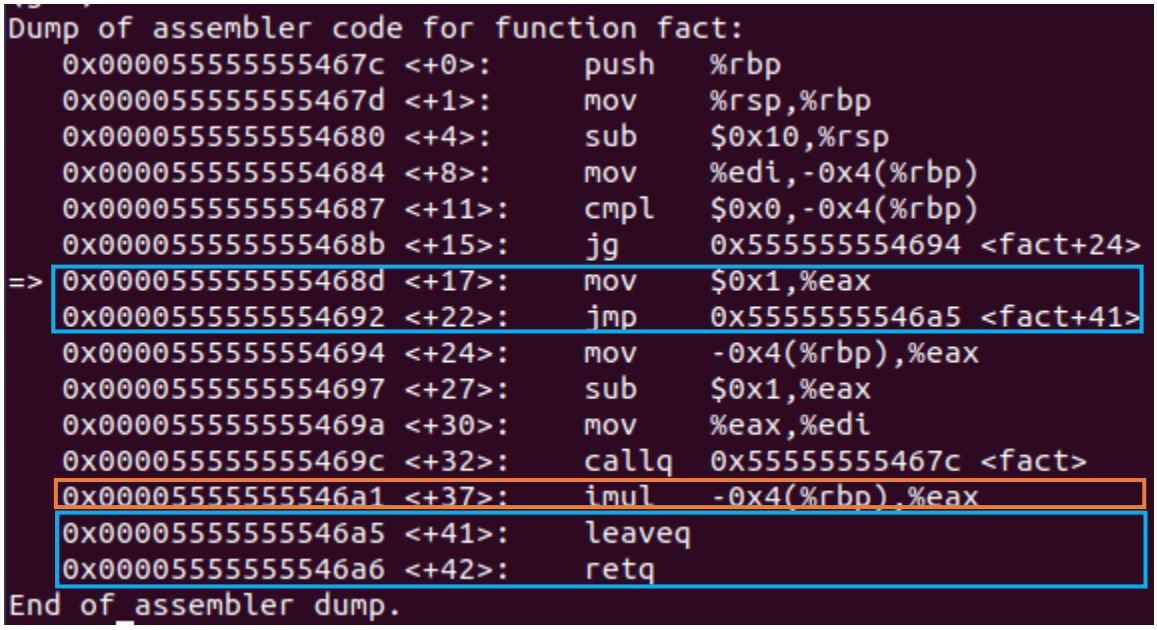

Figure 17: Assembler Code of fact() after three calls

Since no jump occurs, the next instructions in blue first value 0x01 is moved to register `$eax`. Then a jump will occur to the instruction at address 0x5555555546A5 which will terminate the fourth call. The `leaveq` instruction will revert the stack frame to its previous state prior to the fourth function call by restoring the stack and base pointers. Also, `retq` will jump back to the return address stored on top of the stack. Hence, the next instruction to be executed (in orange) is at address 0x5555555546A1 (i.e., the return address). Figure 18 shows the reverted stack frame (prior to the fourth call). The return address (in amber) will also be popped off the stack. Note that figure 18 below is the stack frame as it was shown in figure 12.

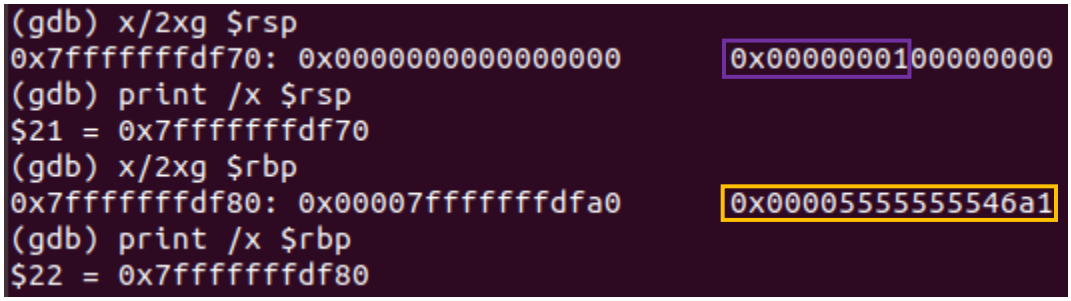

Figure 18: Reverted stack frame prior to fourth call

The value of the argument (in purple) in this stack frame is 0x01. This is multiplied by value contained in register `$eax` (i.e., 0x01) where the result is stored back into `$eax`.

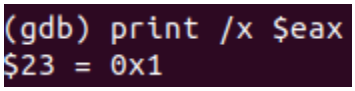

Figure 19: Value stored in register `$eax` after multiplication

Return to Second Call

After the execution of the `imul` instruction, we return back to the set of instructions in blue. Now, The `leaveq` instruction will revert the stack frame to its previous state prior to the third function call by restoring the stack and base pointers. Also, `retq` will jump back to the return address stored on top of the stack. Again, the next instruction to be executed (in orange) is at the return address. Figure 20 shows the reverted stack frame (prior to the third call). The return address (in amber) will also be popped off the stack. Note that figure 20 below is the stack frame as it was shown in figure 9.

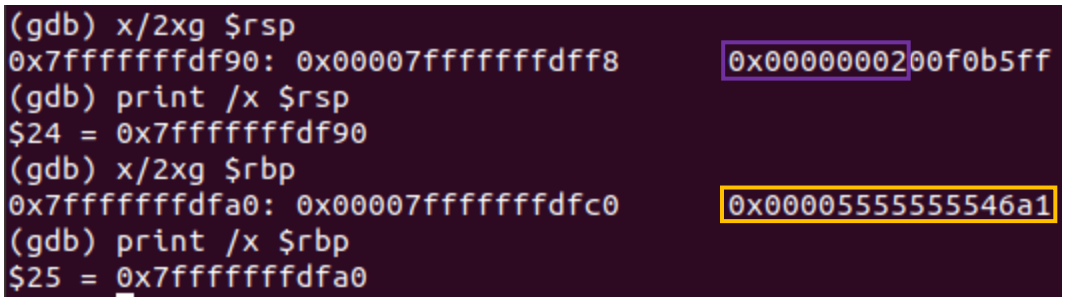

Figure 20: Reverted stack frame prior to third call (return to second call)

The value of the argument (in purple) in this stack frame is 0x02. This is multiplied by value contained in register `$eax` (i.e., 0x01) where the result is stored back into `$eax`.

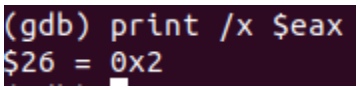

Figure 21: Value stored in register `$eax` after multiplication

Return to First Call

Again, we return to the set of instructions in blue and prepare to revert the stack frame prior to the second call. The next instruction to be executed (in orange) is at the return address. Figure 22 shows the reverted stack frame (prior to the third call). The return address (in amber) will also be popped off the stack. Note that figure 22 below is the stack frame as it was shown in figure 6.

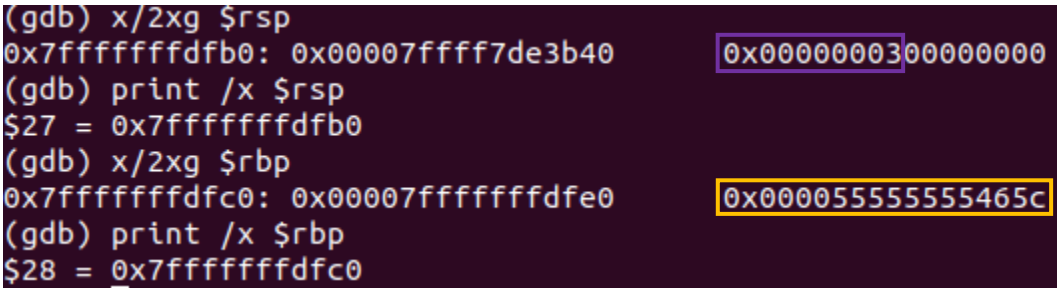

Figure 22: Reverted stack frame prior to second call (return to first call)

The value of the argument (in purple) in this stack frame is 0x03. This is multiplied by value contained in register `$eax` (i.e., 0x02) where the result is stored back into `$eax`. At this point, we have obtained the final return value of `fact(3)` function (i.e., 3! = 6).

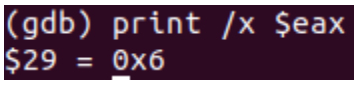

Figure 23: Value stored in register `$eax` after multiplication

The next instruction to be executed will be the return address 0x55555555465C, which is located in `main()`.

Return to Main

```
Dump of assembler code for function main:
   0x000055555555464a <+0>:     push   %rbp
   0x000055555555464b <+1>:     mov    %rsp,%rbp
   0x000055555555464e <+4>:     sub    $0x10,%rsp
   0x0000555555554652 <+8>:     mov    $0x3,%edi
   0x0000555555554657 <+13>:    callq  0x55555555467c <fact>
=> 0x000055555555465c <+18>:    mov    %eax,-0x4(%rbp)
   0x000055555555465f <+21>:    mov    -0x4(%rbp),%eax
   0x0000555555554662 <+24>:    mov    %eax,%esi
   0x0000555555554664 <+26>:    lea    0xc9(%rip),%rdi        # 0x555555554734
   0x000055555555466b <+33>:    mov    $0x0,%eax
   0x0000555555554670 <+38>:    callq  0x555555554520 <printf@plt>
   0x0000555555554675 <+43>:    mov    $0x0,%eax
   0x000055555555467a <+48>:    leaveq
   0x000055555555467b <+49>:    retq
```

Figure 24: Assembler Code of `main()` after returning from `fact()`

We've returned back to the `main()` function and starting where we left off (after the `callq` instruction). Figure 25 shows the reverted stack frame as it was in figure 3.

```
(gdb) x/2xg $rsp
0x7fffffffdfd0: 0x00007fffffffe0c0      0x0000000000000000
(gdb) print /x $rsp
$30 = 0x7fffffffdfd0
(gdb) x/2xg $rbp
0x7fffffffdfe0: 0x00005555555546b0      0x00007ffff7a03bf7
(gdb) print /x $rbp
$31 = 0x7fffffffdfe0
```

Figure 25: Reverted stack frame (return to main)

The last crucial instruction to be executed (in blue) is to move the value stored in register `$eax` (i.e., the return value of `fact()`) to local variable `a`.

```
(gdb) x $rbp - 0x4
0x7fffffffdfdc: 0x00000006
(gdb) x &a
0x7fffffffdfdc: 0x00000006
```

Figure 26: Location and value of local variable `a` on stack

The following instructions in white are responsible for printing the value of this local variable to the console. Since printing to the console is not necessary to run the program, these instructions will not be analyzed here. The reader is welcomed to investigate the `print()` function call as a further exercise. Upon the execution of all instructions, all previous stack frames are destroyed and the first stack frame is reverted to its original state prior to running the program.