CS2002 C Architecture Practical Report

220019540

Tutor: Stephen McQuistin

Overview

In this practical, I was tasked with implementing the necessary functions for a StackFrame module in C.

The first task was to examine the assembly code file that is generated from running Factorial.c, and commenting + explaining the purpose of each line in the code in order to make it readable for a human.

My next task was to then implement the necessary functions to create a StackFrame module in order to provide insight into the structure and state of the call stack at runtime. It allows you to inspect the stack frames, which are data structures containing information about the function calls made by your program up to the point where the code is executed. This can be particularly useful for debugging to see the state of the stack and understand the sequence of function calls.

Afterwards, I analysed the output of my StackFrame code and tracked down what each address corresponds to in my code and how the code works on a low-level programming basis

Assembly Code Examination

The first section of this assignment involved examining the assembly code from the factorial function in Factorial.c and commenting out the purpose and functionality of each line to make it readable for any humans looking at the code.

Here is the snippet of my comments for the factorial function (full thing can be found in `Factorial-Commented-220019540.s`):

```
factorial:
.LFB0:
    .cfi_startproc
    pushq %rbp                          # Save the current base pointer
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq %rsp, %rbp                     # Set the base pointer to the current stack pointer.
    .cfi_def_cfa_register 6
    subq $16, %rsp                      # Allocate space for local variables
    movq %rdi, -8(%rbp)                 # Store the first argument 'n' in the local variable
    movq %rsi, -16(%rbp)                # Store the second argument 'accumulator' in the local variable
    cmpq $1, -8(%rbp)                   # Compare 'n' with 1 to check for the base case
    ja .L2                              # If 'n' is greater than 1, jump to label .L2 (recursive case)
    movl $7, %edi                       # Load 7 into %edi, preparing to call printStackFrames
    call printStackFrames               # Call printStackFrames function to print stack frame data
    movq -16(%rbp), %rax                # Move 'accumulator' into return register %rax
    jmp .L3                             # Jump to label .L3 to exit function
.L2:
    movq -8(%rbp), %rax                 # Load 'n' into %rax
    imulq -16(%rbp), %rax               # Multiply 'n' by 'accumulator', result in %rax
    movq -8(%rbp), %rdx                 # Load 'n' into %rdx
    subq $1, %rdx                       # Decrement 'n' by 1
    movq %rax, %rsi                     # Set up second argument for recursive call: updated 'accumulator'
    movq %rdx, %rdi                     # Set up first argument for recursive call: decremented 'n'
    call factorial                      # Recursive call to factorial function
.L3:
    leave                               # Restore previous stack frame
    .cfi_def_cfa 7, 8
    ret                                 # Return to caller
    .cfi_endproc
.LFE0:
    .size factorial, .-factorial        # Specify the size of the factorial function
    .section .rodata                    # Begin the read-only data section for string literals
    .align 8                            # Align the following data on an 8-byte boundary
.LC0:
    .string "executeFactorial: basePointer = %lx\n" # String literal for printing the base pointer
    .align 8                            # Align the following data on an 8-byte boundary
.LC1:
    .string "executeFactorial: returnAddress = %lx\n" # String literal for printing the return address
    .align 8                             # Align the following data on an 8-byte boundary
.LC2:
    .string "executeFactorial: about to call factorial which should print the stack\n" # String literal
indicating the call to factorial
    .align 8                             # Align the following data on an 8-byte boundary
.LC3:
    .string "executeFactorial: factorial(%lu) = %lu\n" # String literal for printing the result of
factorial
```

```
.text                                   # Switch back to the text section for code
.globl executeFactorial                 # Declare executeFactorial as a global symbol
.type executeFactorial, @function       # Specify the type of executeFactorial as a function
```

How does this correspond to the C source code?

In the assembly code, the `factorial` function begins by establishing a stack frame, which is a section of the stack dedicated to handling the local variables and return information for that particular function call. This is achieved by pushing the current base pointer (`%rbp`) onto the stack and then setting the base pointer to the current stack pointer (`%rsp`). The stack pointer is then decremented to reserve space for local variables (`subq $16, %rsp`), which corresponds to the local variables in the C code: the parameters `n` and `accumulator`.

In the C code, these local variables are the parameters of the `factorial` function itself, and they are used directly within the function. The assembly code, however, must manually store these parameters into the local stack space it has created. It does so by moving the first (`%rdi`) and second (`%rsi`) arguments into the designated local variable space at `-8(%rbp)` and `-16(%rbp)`, respectively. These moves correlate to the assignment of `n` and `accumulator` in the C function's parameter list.

The comparison (`cmpq $1, -8(%rbp)`) and conditional jump (`ja .L2`) instructions in the assembly implement the `if` statement in C, determining whether the base case of the recursion (`n <= 1`) has been reached. If the base case is true, the assembly code calls the `printStackFrames` function and then prepares to return the accumulator by moving its value into the `%rax` register, which is the designated return register in x86-64 assembly.

For the recursive case, the assembly code mimics the C code's recursive call to `factorial(n - 1, n * accumulator)` by first performing the arithmetic operations on `n` and `accumulator` and then setting up the arguments for the next `factorial` call. The result of `n * accumulator` is placed in `%rax`, which is then moved to `%rsi` as the second argument for the recursive call. The decremented `n` is placed in `%rdi` as the first argument. Then, `call factorial` initiates the recursive call.

Finally, the function concludes by restoring the previous stack frame (`leave`) and returning control to the caller (`ret`), which matches the C code's `return` statement.

Implementation

The next task in this practical was to create an implementation of the functions found in StackFrame.c.

(write a little more here: what was the essential part of implementation, struggles, learning experiences etc.

Function explanation

Below is an explanation of each function in my code and how it operates with regards to the StackFrame:

`getBasePointer()`:

This function retrieves the current base pointer (frame pointer), which points to the start of the current stack frame. The base pointer is typically stored in the `rbp` register on x86_64 systems. The function uses inline assembly to move the value from the `rbp` register into the `rax` register, which is then returned to the caller. The base pointer is essential for navigating the stack because it provides a reference point for accessing function parameters, local variables, and the previous stack frame's base pointer.

`getReturnAddress()`:

Similar to `getBasePointer`, this function uses inline assembly to access the return address of the current function. The return address is stored on the stack and is the address to which the program will jump back once the current function completes. The function moves the value at the memory location immediately above the current base pointer (which is the return address) into the `rax` register and returns it. This address is crucial for understanding the program's control flow, as it indicates where the execution should continue after the function returns.

`printStackFrameData()`:

This function prints the contents of a stack frame. It takes two arguments: a pointer to the base of the current stack frame and a pointer to the base of the previous stack frame. It uses a loop to iterate over the memory addresses between these two pointers, shifting and masking operations to extract individual bytes, and then calls `printf` to print each byte in hexadecimal format. This function is useful for inspecting the values stored on the stack, which may include local variables, saved registers, and possibly return addresses from previous function calls.

`printStackFrames()`:

This function is responsible for printing a specified number of stack frames. It first retrieves the current base pointer using `getBasePointer`. Then, it enters a loop where it prints the current stack frame's data using `printStackFrameData`, retrieves the base pointer of the previous stack frame (which is stored at the current base pointer's location), and updates the loop control variables. The loop continues until the specified number of stack frames has been printed or until it encounters a base pointer with a null value, indicating the end of the stack frames. This function provides a way to traverse

and inspect multiple stack frames, which can be useful for debugging or understanding the call stack's state at a particular point in the program's execution.
Each of these methods in `StackFrame.c` uses low-level operations and inline assembly to provide insights into the runtime stack's structure and contents, which are otherwise abstracted away in higher-level programming.

In practice, when this code is executed, it will output a section of the program's call stack, showing return addresses and the data present in the stack frames. This can help developers to trace back the sequence of function calls that led to the current point of execution, and to inspect the data that each function is working with on the stack. This can be helpful for finding issues related to the stack, such as buffer overflows, stack corruption, or simply to understand the program flow up to a certain point.

Testing

For testing my program, I decided to verify the correct procedures of my code by comparing the output from running objdump and the output from my own program. I also added print statements to test individual methods and see if they correctly work on their own.

Verifying the correct return address

I compared the output from objdump and the return address that my program retrieved to verify that the correct address was displayed. My program listed the caller return address as 401154 - this was achieved by moving the value of %rbp into a different register (in this case %rax) and dereferencing it then offsetting the value by 8 to get the caller return address. In the objdump output we see the address 401154 in the .main section:

```
0000000000401146 <main>:
  401146:       55                      push   %rbp
  401147:       48 89 e5                mov    %rsp,%rbp
  40114a:       b8 00 00 00 00          mov    $0x0,%eax
  40114f:       e8 4c 00 00 00          callq  4011a0
<executeFactorial>
  401154:       b8 00 00 00 00          mov    $0x0,%eax
  401159:       5d                      pop    %rbp
  40115a:       c3                      retq
```

The return address `401154` matches the address after the call to `executeFactorial` in the `main` function, indicating where the program will return after `executeFactorial` is done. This verifies that the program is correctly retrieving the right return address fo the caller.

Trying a different value of n

I tried a different value of n in the definition of DEFAULT_VALUE_OF_N to see if my program would still work:

```
/*
 * The default number to use when computing factorial.
 */
#define DEFAULT_VALUE_OF_N 8l
```

I received the following output:

```
aw356@lyrane:.../Documents/cs2002/W08-C-Architecture/src $
./TryStackFrames
executeFactorial: basePointer = 7ffdfd175610
executeFactorial: returnAddress = 401154
executeFactorial: about to call factorial which should print the stack

-------------
00007ffdfd1754d0: 00007ffdfd1754f0 --  f0 54 17 fd fd 7f 00 00
00007ffdfd1754d8: 000000000040117c --  7c 11 40 00 00 00 00 00
00007ffdfd1754e0: 0000000000009d80 --  80 9d 00 00 00 00 00 00
00007ffdfd1754e8: 0000000000000001 --  01 00 00 00 00 00 00 00
-------------
00007ffdfd1754f0: 00007ffdfd175510 --  10 55 17 fd fd 7f 00 00
00007ffdfd1754f8: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffdfd175500: 0000000000004ec0 --  c0 4e 00 00 00 00 00 00
00007ffdfd175508: 0000000000000002 --  02 00 00 00 00 00 00 00
-------------
00007ffdfd175510: 00007ffdfd175530 --  30 55 17 fd fd 7f 00 00
00007ffdfd175518: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffdfd175520: 0000000000001a40 --  40 1a 00 00 00 00 00 00
00007ffdfd175528: 0000000000000003 --  03 00 00 00 00 00 00 00
-------------
00007ffdfd175530: 00007ffdfd175550 --  50 55 17 fd fd 7f 00 00
00007ffdfd175538: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffdfd175540: 0000000000000690 --  90 06 00 00 00 00 00 00
00007ffdfd175548: 0000000000000004 --  04 00 00 00 00 00 00 00
-------------
00007ffdfd175550: 00007ffdfd175570 --  70 55 17 fd fd 7f 00 00
```

```
00007ffdfd175558: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffdfd175560: 0000000000000150 --  50 01 00 00 00 00 00 00
00007ffdfd175568: 0000000000000005 --  05 00 00 00 00 00 00 00
-------------
00007ffdfd175570: 00007ffdfd175590 --  90 55 17 fd fd 7f 00 00
00007ffdfd175578: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffdfd175580: 0000000000000038 --  38 00 00 00 00 00 00 00
00007ffdfd175588: 0000000000000006 --  06 00 00 00 00 00 00 00
-------------
00007ffdfd175590: 00007ffdfd1755b0 --  b0 55 17 fd fd 7f 00 00
00007ffdfd175598: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffdfd1755a0: 0000000000000008 --  08 00 00 00 00 00 00 00
00007ffdfd1755a8: 0000000000000007 --  07 00 00 00 00 00 00 00
-------------
00007ffdfd1755b0: 00007ffdfd1755d0 --  d0 55 17 fd fd 7f 00 00
00007ffdfd1755b8: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffdfd1755c0: 0000000000000001 --  01 00 00 00 00 00 00 00
00007ffdfd1755c8: 0000000000000008 --  08 00 00 00 00 00 00 00
-------------
00007ffdfd1755d0: 00007ffdfd175610 --  10 56 17 fd fd 7f 00 00
00007ffdfd1755d8: 0000000000401225 --  25 12 40 00 00 00 00 00
00007ffdfd1755e0: 0000000000000040 --  40 00 00 00 00 00 00 00
00007ffdfd1755e8: 0000000000000001 --  01 00 00 00 00 00 00 00
00007ffdfd1755f0: 0000000000000008 --  08 00 00 00 00 00 00 00
00007ffdfd1755f8: 0000000000000000 --  00 00 00 00 00 00 00 00
00007ffdfd175600: 0000000000401154 --  54 11 40 00 00 00 00 00
00007ffdfd175608: 00007ffdfd175610 --  10 56 17 fd fd 7f 00 00
executeFactorial: factorial(8) = 40320
```

The value of 8 factorial is indeed 40320, which verified that my program was able to work with different values of n.

Method Testing

I also added print statements on all my methods to verify that they are working correctly in my program:

```
unsigned long getBasePointer() {
    unsigned long basePointer;
```

```
    asm("movq (%%rbp), %0" : "=r" (basePointer));
    printf("Test getBasePointer: basePointer = %lx\n", basePointer); // Test
    return basePointer;
}
```

2.

```
unsigned long getReturnAddress() {
    unsigned long returnAddress;
    (...)
    printf("Test getReturnAddress: returnAddress = %lx\n", returnAddress);
    return returnAddress;
}
```

3.

```
void printStackFrameData(unsigned long basePointer, unsigned long
previousBasePointer) {
    printf("Test printStackFrameData: Printing stack frame data from %lx to %lx\n",
basePointer, previousBasePointer); // Test print statement
}
```

4.

```
void printStackFrames(int number) {
    printf("Test printStackFrames: Starting to print %d stack frames\n", number); //
Test print statement
    // ... rest of the function ...
    printf("Test printStackFrames: Finished printing stack frames\n"); // Test print
statement
}
```

Here is the output verifying that my program returns the correct values in my program:

```
aw356@lyrane:.../Documents/cs2002/W08-C-Architecture/src $ ./TryStackFrames
Test getBasePointer: basePointer = 7ffc9075b3e0
executeFactorial: basePointer = 7ffc9075b3e0
```

```
Test getReturnAddress: returnAddress = 401154
executeFactorial: returnAddress = 401154
executeFactorial: about to call factorial which should print the stack

Test printStackFrames: Starting to print 7 stack frames
Test getBasePointer: basePointer = 7ffc9075b2e0
-------------
Test printStackFrameData: Printing stack frame data from 7ffc9075b2e0 to 7ffc9075b300
00007ffc9075b2e0: 00007ffc9075b300 --  00 b3 75 90 fc 7f 00 00
00007ffc9075b2e8: 000000000040117c --  7c 11 40 00 00 00 00 00
00007ffc9075b2f0: 00000000000002d0 --  d0 02 00 00 00 00 00 00
00007ffc9075b2f8: 0000000000000001 --  01 00 00 00 00 00 00 00
Test printStackFrames: Finished printing stack frames
-------------
Test printStackFrameData: Printing stack frame data from 7ffc9075b300 to 7ffc9075b320
00007ffc9075b300: 00007ffc9075b320 --  20 b3 75 90 fc 7f 00 00
00007ffc9075b308: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffc9075b310: 0000000000000168 --  68 01 00 00 00 00 00 00
00007ffc9075b318: 0000000000000002 --  02 00 00 00 00 00 00 00
Test printStackFrames: Finished printing stack frames
-------------
Test printStackFrameData: Printing stack frame data from 7ffc9075b320 to 7ffc9075b340
00007ffc9075b320: 00007ffc9075b340 --  40 b3 75 90 fc 7f 00 00
00007ffc9075b328: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffc9075b330: 0000000000000078 --  78 00 00 00 00 00 00 00
00007ffc9075b338: 0000000000000003 --  03 00 00 00 00 00 00 00
Test printStackFrames: Finished printing stack frames
-------------
Test printStackFrameData: Printing stack frame data from 7ffc9075b340 to 7ffc9075b360
00007ffc9075b340: 00007ffc9075b360 --  60 b3 75 90 fc 7f 00 00
00007ffc9075b348: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffc9075b350: 000000000000001e --  1e 00 00 00 00 00 00 00
00007ffc9075b358: 0000000000000004 --  04 00 00 00 00 00 00 00
Test printStackFrames: Finished printing stack frames
-------------
Test printStackFrameData: Printing stack frame data from 7ffc9075b360 to 7ffc9075b380
00007ffc9075b360: 00007ffc9075b380 --  80 b3 75 90 fc 7f 00 00
00007ffc9075b368: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffc9075b370: 0000000000000006 --  06 00 00 00 00 00 00 00
00007ffc9075b378: 0000000000000005 --  05 00 00 00 00 00 00 00
Test printStackFrames: Finished printing stack frames
-------------
Test printStackFrameData: Printing stack frame data from 7ffc9075b380 to 7ffc9075b3a0
00007ffc9075b380: 00007ffc9075b3a0 --  a0 b3 75 90 fc 7f 00 00
00007ffc9075b388: 000000000040119e --  9e 11 40 00 00 00 00 00
00007ffc9075b390: 0000000000000001 --  01 00 00 00 00 00 00 00
00007ffc9075b398: 0000000000000006 --  06 00 00 00 00 00 00 00
Test printStackFrames: Finished printing stack frames
```

```
-------------
Test printStackFrameData: Printing stack frame data from 7ffc9075b3a0 to 7ffc9075b3e0
00007ffc9075b3a0: 00007ffc9075b3e0 --  e0 b3 75 90 fc 7f 00 00
00007ffc9075b3a8: 0000000000401225 --  25 12 40 00 00 00 00 00
00007ffc9075b3b0: 0000000000000040 --  40 00 00 00 00 00 00 00
00007ffc9075b3b8: 0000000000000001 --  01 00 00 00 00 00 00 00
00007ffc9075b3c0: 0000000000000006 --  06 00 00 00 00 00 00 00
00007ffc9075b3c8: 0000000000000000 --  00 00 00 00 00 00 00 00
00007ffc9075b3d0: 0000000000401154 --  54 11 40 00 00 00 00 00
00007ffc9075b3d8: 00007ffc9075b3e0 --  e0 b3 75 90 fc 7f 00 00
Test printStackFrames: Finished printing stack frames
executeFactorial: factorial(6) = 720
```

Output

In order to verify that my program is working correctly, I will be examining the output of my own code and explaining how it works according to the C code of StackFrame and Factorial:

Comparing program output to source code

When `executeFactorial` is called, it first retrieves the base pointer and return address of the current stack frame, which are then printed to the console. These values correspond to the memory address of the base of the stack frame and the instruction to return to once the function completes, respectively. The output shows the base pointer as `7ffe34877400` and the return address as `401154`, which is the address in the `main` function where `executeFactorial` is called.

As `executeFactorial` announces its intent to call the `factorial` function, the program output then displays the stack contents, with memory addresses and their corresponding values. These stack contents include return addresses, such as `40119e`, which is likely the address within the `factorial` function where it calls itself recursively, and other data like the current value of the factorial calculation and loop counters.

The `printStackFrames` function, called when the base case of the recursion (`n <= 1`) is reached, outputs the stack frame data, showing how the stack evolves with each recursive call. The program output lists several memory addresses with their contents, which include pointers to the next stack frame and return addresses. The final output line, `executeFactorial: factorial(6) = 720`, shows the result of the factorial calculation, which is the culmination of the recursive calls made by the `factorial` function.

In the disassembled `objdump` output, we can see the assembly instructions that correspond to these high-level operations. For example, the `callq` instruction at `4011ad` corresponds to the call to `getBasePointer`, and the `callq` at `4011d1` corresponds to the call to `getReturnAddress`. The `callq` instruction at `401177` within the `factorial` function corresponds to the call to `printStackFrames`. These assembly instructions are the low-level representations of the source code functions, and their execution leads to the program output observed.

Comparing objdump output to program output

I ran my program and objdump and stored the output from both in a file *output.txt* to analyse and compare both.

I looked at the output of running objdump on my code and compared it to the output of my program to verify that the correct values are being printed:

Base Pointer and Return Address:

In `executeFactorial`, the base pointer and return address are retrieved and printed.
The program output shows:

```
executeFactorial: basePointer = 7ffe34877400
executeFactorial: returnAddress = 401154
```

The corresponding assembly in `executeFactorial` (objdump output) for getting the base pointer is at `4011ad` with the call to `getBasePointer` and for the return address at `4011d1` with the call to `getReturnAddress`.
The return address `401154` matches the address after the call to `executeFactorial` in the `main` function, indicating where the program will return after `executeFactorial` is done.

The `printStackFrames` function is called within `factorial` when `n <= 1`.
The program output shows the stack being printed with addresses and values.
The objdump output shows the call to `printStackFrames` at `401177` within the `factorial` function.

The `factorial` function is recursive, and the program output shows the result of `factorial(6) = 720`.
The objdump output shows the recursive call to `factorial` at `401199`.

Stack Frame Data:

The program output includes stack data such as:

```
00007ffe348773f0: 0000000000401154 --  54 11 40 00 00 00 00 00
00007ffe348773f8: 00007ffe34877400 --  00 74 87 34 fe 7f 00 00
```

These lines show the return address and base pointer for a stack frame.
The return address `401154` matches the address in the `main` function after the call to `executeFactorial`, and the base pointer `7ffe34877400` matches the base pointer printed by `executeFactorial`.

The `callq` instruction in the objdump output is used to call functions. For example, `callq 4011a0 <executeFactorial>` in the `main` function calls `executeFactorial`.
The `mov` instructions are used to move data around, such as moving return values into the right place for the next operation.

The return addresses and base pointers in the program output directly correspond to the addresses seen in the objdump output, showing the execution of the program and the structure of the stack frames.

Evaluation

I think my program has successfully achieved every requirement detailed in the specification of the coursework. My program is able to print stackframes displaying memory addresses, pointers and numbers in the correct format and representation and works correctly, as shown when comparing values between the program output and objdump. After making use of features such as inline assembly and inspecting the call stack at a much more detailed level, I was able to achieve a working solution that can display the current processes of the CPU at runtime of the program.

Conclusion

I greatly enjoyed working on this coursework and learning how to work with assembly and runtime processor memory to get a more detailed look into all the processes that happen at a low level in the processor and the registers associated with it. If given more time, I would like to explore working with stack frames on different types of program, and see how the stack frame would look like in one of those instances.