Anirudh Tunoori

netid: at813

RUID: 165003175

pa3: mystery

What does the mystery do?

The mystery program reads an integer n and prints/outputs the nth fibonacci number in the (self-generated) fibonacci sequence. I determined the program function by first tracing the assembly code in mystery.c. I realized/thought that it was taking an integer and applied it to some formula to produce some result. The formula was applied in a recursive process. I then built an executable of the assembly code and ran a few values (0-9) in the function. I was already familiar with the fibonacci sequence and so I noticed the pattern of adding the two preceding numbers to determine the nth number.

Implementation:

My formula program consisted of 3 files: mystery.c, mystery.h, and the makefile. My makefile contained 3 rules, mystery which built the mystery executable, mystery which built mystery.o, and clean. The mystery.s file had three functions an add function a “dothething” function and a main function. I created the function prototypes of the add and “dothething” functions in the mystery.h file which i determined from tracing mystery.s. I noticed that the add function was just adding two ints so I implemented that function first. I knew that dothething can find the nth fibonacci by determining n-1 and n-2 in the sequence and that led me to realize that this would be my recursive case. (add(dothething(n-2),dothething(n-1))). I was not able to realize what the base case to this function was however so I decided to come back to it later (n =0 or n = 1 was what I thought). I also was unsure how the previous numbers in the sequence could be recalled. Closer examination told me that some kind of data structure was being used which was called “num”. I used trial and error to determine that i was right int thinking that num was an int array. The challenge was determining what the size of the array was and where it was being initialized since i believed that those two factors can change the assembly code significantly. I went to the original mystery.s and I was confused because anything greater than 46(inclusive) gave an error so I assumed that that was the size of the array. Then I noticed that this number was not present in mystery.s on the other hand I found a 199 in the instruction that contained (jle 199) which led me to believe that the size must be 200, and that the array would have to be declared outside the functions and accessed by both. I then realized that the main function was essentially filling this array with negative 1's and I knew that I would therefore have (in the dothething function) an if statement with two base cases followed by the recursive case inside. I did not encounter any other challenges in generating the remainder of the C Code and I made sure to follow the variable names. I then built my own mystery.s (optimized and unoptimized) and examined them, Other than a few mild changes I found that my mystery.s and the mystery.s provided to me very similar in all of the non-trivial areas. There were a few "stylistic" differences in the code but the functions were the same.

This program uses an in array data structures and a few other primitive int datatypes (2^31 -1). The program is solely reliable for/adept at computing all of the fibonacci numbers up to 46(inclusive).

Optimization:The optimized assembly was better at allocating the required space for the data, functions, and etc. (fewer bytes). The stark difference was in the number of lines that were reduced in the optimized assembly i.e: start and end proc were used for function entry and exit in place of a total 5-6 instructions. The unoptimized code overused "cfi. cfa…offset…. and reset" and about a third of the code (main operations/processes) were the same. There were ten or twelve lines where the operands and registers were the same but had subtle changes. Finally I felt that the optimized code did better with the recursive calls (there were about three instructions before each dothething function call) and using/storing/accessing the array (num(,%ebx,4),num(, %edx,4)).

Runtime Analysis: There were three functions involved in this program. The main function essentially fills an int array of size 200 with negative 1 so that is O(200) -> O(1). The add function is called O(n) times by the dothething function but runs at O(1) (a single addition operation), which leaves the bulk of the runtime to be taken up by the dothething function. It makes O(n) calls each for the (n-1) and (n-2) which approximates to O(2n) ->O(n). The array storage and the recursive design results in a relatively feasible overall runtime.

Space Analysis: The program can only take an int (2^31 -1)as an input. Anything larger than 46 will result in an overflow (this is because pointers are freed beyond 46). Negative numbers will also produce an error (not accounted for by mystery.c) The bulk of the space is taken up by the int array of size 200 (800 bytes).