Low-Level Programming Language Course Project - CSCI 3371

Tim Price
Augusta University
Email: timprice@augusta.edu

I. Introduction

This projects objective was to compare the implementation of a stack-based postfix expression evaluator written in both C# and 32-bit MASM assembly. This project also includes an evaluation of both programs performance with respect to throughput and latency.

II. Project Components

A. Postfix Evaluation

For a detailed writing on how postfix evaluation works see here. The basic idea is the the arithmetic operator follows the numbers. For example given the expression ”63*” we see digits 6 and 3 followed by ”*”. So this is $6 * 3 = 18$. We can implement this with a stack data-structure by pushing all digits to the stack and popping the last two added when we see an operator such as ”*”.

B. C# Implementation

```
using System;
using System.Collections.Generic;

class PostfixEvaluation {
    static int EvaluatePostfix(string expression) {
        Stack<int> stack = new Stack<int>();
        foreach (char c in expression) {
            if(char.IsDigit(c)) {
                stack.Push(c - '0');
                /* char -> int -> push to stack */
            } else if (c == ' '){
                continue;
            } else {
                int a = stack.Pop();
                int b = stack.Pop();
                switch(c) {
                    case '+':
                        stack.Push(a + b);
                        break;
                    case '-':
                        stack.Push(b - a);
                        break;
                    case '*':
                        stack.Push(a * b);
                        break;
                    case '/':
                        stack.Push(b / a);
                        break;
                }
            }
        }
        return stack.Pop();
    }
    static void Main(){
        string expression = "231*+9 - ";
        Console.WriteLine("Postfix Evaluation : " + EvaluatePostfix(expression));
    }
}
```

C. C# Implementation Notes

In essence, this program scans a line of text (a postfix expression; to be evaluated), reading each character one-at-a-time. If the character is a number (0-9) then it gets converted from a ’char’ type to an ’int’ and pushed onto the stack. If the character is one of ”+ - * /” than the program pops the 2 most recently pushed numbers off of the stack, completes the operation (whichever operator was detected) with the 2 popped numbers, and pushes that result onto the stack. The program looks through each character in the expression until it reaches the end of the string, then pops the last integer off the stack which should be the final result of all the evaluation.

D. Assembly Implementation

```
    .386
.model flat, stdcall
.stack 4096

ExitProcess PROTO, dwExitCode:DWORD

.data
    ; expression BYTE "231*+9-", 0
    expression BYTE "92*", 0
    the_stack DWORD 25 dup(?)
    result DWORD 0
    stack_point DWORD 0

.code
    main PROC
        mov esi, offset expression

    READ_STRING:
        movzx eax, byte ptr [esi]
        cmp al, 0
        je END_LOOP

        cmp al, '0'
        jl HANDLE_OPERATOR
```

```
        cmp al, ’9’
        jg HANDLE_OPERATOR

        sub al, ’0’     ; ASCII conversion
        movzx eax, al
        mov ecx, stack_point
        mov [the_stack + ecx*4], eax
        inc stack_point
        inc esi
        jmp READ_STRING

    HANDLE_OPERATOR:
        dec stack_point
        mov ecx, stack_point
        mov eax, [the_stack + ecx*4] ; popped
        dec stack_point
        mov ecx, stack_point
        mov ebx, [the_stack + ecx*4] ; popped

        cmp byte ptr [esi], ’+’
        je DO_ADD
        cmp byte ptr [esi], ’-’
        je DO_SUB
        cmp byte ptr [esi], ’*’
        je DO_MUL
        jmp INVALID_OP

    INVALID_OP:
        jmp END_LOOP

    DO_ADD:
        add ebx, eax
        jmp PUSH_RESULT

    DO_SUB:
        sub ebx, eax
        jmp PUSH_RESULT

    DO_MUL:
        imul ebx, eax
        jmp PUSH_RESULT

    PUSH_RESULT:
        mov ecx, stack_point
        mov [the_stack + ecx*4], ebx
        inc stack_point
        inc esi
        jmp READ_STRING

    END_LOOP:
        dec stack_point
        mov ecx, stack_point
        mov eax, [the_stack + ecx*4]
        mov result, eax

        INVOKE ExitProcess, 0
    main ENDP

END main
```

E. Assembly Implementation Notes

The program accomplished the same task as the C# program but in assembly, meaning translation is quite verbose, all though they do share similar structure. Some of the major differences include how the program evaluates the character and how the looping mechanism works.

For example, to determine which character the pointer is looking at, we have to consider that it is currently in ASCII format because it is from a string. We go from ASCII format to hexadecimal through the ”sub al, ’0’” instruction which is similar to what I used in the C# conversion.

Up to this point I have 2 programs that evaluate a postfix expression. The C# implementation prints to console prints an signed integer result. The Assembly program stores its result in memory and in register EAX. You can view the result at this point by creating a break-point in a debugger just before the program exits. In the next section we will be improving this system.

F. Improving Assembly with dependencies

This section is largely based in Chapter 11 in the ”Assembly Language for x86 Processors, 8th edition” (see here) used throughout this course. This chapter details using the windows operating system AP and MASM software-development-kit as dependencies to extend the capabilities of MASM programs. The goal here is to print our result to the shell and prepare the program to use performance evaluation tooling. In section 11.1.5 The author shows an example of using ”ConsoleWrite” from the windows API to write sections of memory into the shell. I also read and utilized ideas from an old blog on Input/Output in MASM see here taught me enough to write this which just adds an extra layer of polish to the program. Places labeled ”some code” existed before these changes, and remain the same.

```
            ; some code

    ; libraries included in MASM SDK
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib

.data
            ; some code

    buffer db 32 dup(0)
    fmtStr db "%d", 0

    ; end of line sequence
    endl EQU <0dh, 0ah>

    consoleHandle DWORD ?
    messageSize DWORD ?
    bytesWritten DWORD ?

.code
    STD_OUTPUT_HANDLE EQU -11

    invoke GetStdHandle, STD_OUTPUT_HANDLE
    mov consoleHandle, eax

            ; some code

    ; Convert result to string
```

```
    invoke wsprintf, addr buffer, addr fmtStr,
         result

    ; Get string length
    invoke lstrlen, addr buffer
    mov messageSize, eax

    ; Write to Console
    INVOKE WriteConsole,
        consoleHandle,
        ADDR buffer,
        messageSize,
        ADDR bytesWritten,
        0

            ; some code
```

As I discussed with our class T.A. getting MASM to write anything to the console is difficult and convoluted. We agreed that this was mostly unnecessary to complete this program, but I was able to get this simple example working after studying the textbook.

I'd like to emphasize that this section of the code requires the kernel and user32 libraries which are included in the masm32 software-development-kit. These dependencies must be linked after assembling this part of the source code. In the project directory I will have included a power-shell script that assembles and links everything together nicely, with instructions in a read-me. Its important to note that the script looks in the default location $C : /masm32$ where the SDK is installed on windows. See here to view or install the SDK.

III. Evaluation: Performance Comparison

For a balanced performance evaluation, I am going to be adding some code to both programs to measure both throughput and latency. Given the nature of assembly programs, I strongly expect that both latency and runtime will be better (faster throughput and less latency) when compared with the C# program. I suspect the C# will be reasonably slower because it must interact with the common language runtime which is a virtual machine similar to the java virtual machine (jvm). I expect there will also be some slowdown as a result of the garbage collector carefully managing memory during program execution.

One possible concern I want to address is that I run Linux natively on all my computers. To succeed in MASM development I run a windows 11 virtual machine. Instead of running my C# code locally I'll of course be executing it on the virtual machine. I'd expect measurements to differ greatly if the programs were to be executed on native windows machines.

A. C# Latency

To measure the latency or "the time it takes to complete one operation" of my C# program I used the stopwatch class from the System Diagnostics package in the C# standard library, see here. To measure throughput I start the timer just before calling the EvaluatePostfix method then stop immediately after. This give some idea of how fast the method is running.

```
class PostfixEvaluation {
    public static Stopwach timer = new
        Stopwatch();

        /* some code /*

    static void Main() {

        /* some code /*

        timer.Restart();
        EvaluatePostfix(expression);
        timer.Stop();
        decimal latency = (decimal)timer.
            Elapsed.TotalMilliseconds;
    }
}
```

I ran this measurement five times and averaged their results which came to roughly **0.0033** milliseconds to run the method once. This was evaluating the given test expression $231{*}{+}9{-}$. I ran the same test with a simpler expression $63*$, which has similar results averaging **0.00256** An important speedup I made was not assigning the return value to a variable. I was originally setting some integer equal to the result of the method which added around 2 milliseconds latency on average.

B. C# Throughput

To measure the throughput I'll take a similar approach to the latency measurement. Throughput is measuring how many times per unit of time something is occurring. $throughput = iterations/time$. I'll be using seconds, and running my program $100,000$ times. This should be enough times to balance out outliers like cache misses, slowdown during program warm-up, etc. So, I'll run the method in a for loop $100,000$ times and divide that by the total time it took to complete all $100,000$ operations.

```
    class PostfixEvaluation {
    public static Stopwach timer = new
        Stopwatch();

        /* some code /*

    static void Main() {

        /* some code /*

        int iterations = 100000;
        timer.Restart();
        for (int i = 0; i < iterations; i++) {
            EvaluatePostfix(expression);
        }
        decimal throughput = iterations / (
            decimal)timer.Elapsed.TotalSeconds
            ;
    }
}
```

When evaluating over 5 separate attempts with the same expressions I did in the latency calculation $231 * +9-$ I get on average **2969897.74** or in other words I could run my method almost 3 million times per second. And for the simpler expression $63*$ run 5 times I unsuprisingly get a higher throughput average of **4522657.972** or roughly 4.5 million operations per second.

C. Latency Assembly

D. Performance Assembly

I tried using API calls to ”QueryPerformanceCounter” but had no luck getting it to work. It measures time in tick rate and the conversion was giving me trouble. The last-second solution I came up with was to just write a loop wrapping around the main procedure of the assembly program, give that a counter starting at 100,000 and continually loop and decrement this counter until it reached zero. I then wrote a basic script in powershell that runs the program while timing it, then calculates throughput and latency and prints them to the shell.

This is what I added to the assembly program which is saved in an additional file $time.asm$.

```

            ; some code

    loopMain PROC
    loopMain ENDP

            ; some code

    main PROC
    mov ecx, loopCounter      ; Set loop count to 100,000
        LOOP_START:
            CALL loopMain
            dec ecx
            jnz LOOP_START
            INVOKE ExitProcess, 0
    main ENDP
```

I timed it using this script

```
$exePath = ".\loop.exe"
$sw = [System.Diagnostics.Stopwatch]::StartNew()
& $exePath
$sw.Stop()

$totalSeconds = $sw.Elapsed.TotalSeconds
$repeatCount = 100000

$throughput = $repeatCount / $totalSeconds
Write-Output "Ran $exePath $repeatCount times"
Write-Output "Total Time: $([math]::Round($totalSeconds, 4)) seconds"
Write-Output "Throughput: $([math]::Round($throughput)) ops/sec"

$exePath = ".\main.exe"
$sw = [System.Diagnostics.Stopwatch]::StartNew()
& $exePath
$sw.Stop()

$totalSeconds = $sw.Elapsed.TotalSeconds
Write-Output "Latency: $([math]::Round($totalSeconds, 4)) seconds"
```

Here I get an average throughput of **0.0104** seconds and an average throughput of **72050** operations per second. These results are quite disappointing. Its difficult to say whether my timekeeping mechanism is causing some lag and skewing my results, maybe the looping mechanism is inefficient, or maybe C# has been very nicely optimized in windows environments. Likely some combination of the three.

IV. Discussion

Like I stated above, I’m disappointed in my performance evaluation results. I very much expect for MASM to be nearly instant especially when compared to C#. If I spent more time to mess with the Windows API and figure out a better solution for timing my MASM code within MASM then I think my results would align better with what I had expected, but that remains unseen. For now I can unconclusively state that my C# program runs quite a bit faster than my MASM implementation.

As far as testing these programs on another machine, the full source code for both programs (including comments and full performance evaluation) were submitted as part of this assignment but can also be found on GitHub here. Inside the directory is organized as C# implementations and Assembly implementations. I have included several assembly files that serve each different purpose. ”main.asm” contains the original assembly source code, ”extra.asm” contains extra API calls to write the result as a string to the console and comes with a power-shell script to assemble and link with the MASM-SDK which is required, and in a sub-directory called timing there is ”time.asm” and its accompanying power-shell script to measure the throughput and latency of my masm program. The C# directory is self explanatory and can be run most easily by using the CLI and executing $dotnetrun$.

Since I worked alone on this project I completed all task/parts of the assignment on my own and will omit the team-contribution section.

V. References

1. Reverse Polish Notation / Postfix Notation Wikipedia https://en.wikipedia.org/wiki/Reverse_Polish_notation

2. Assembly Language for x86 Processors, 8th edition https://asmirvine.com/

3. Krimsky.new MASM I/O Blog http://www.krimsky.net/articles/io_masm.html

4. MASM-32 SDK Installation https://masm32.com/install.htm

5. Microsoft Learn — Stopwatch Class C# https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-9.0