**Rust Programming Lab #6 18th October 2022**

**Accurate Timing**

Even if you are not a serious hacker – concerned to save the last nanosecond in your programme – most of you will need to measure a program (or part of one, e.g. a critical function) run time. Although the crystal clock driving your CPU is usually very accurate – manufacturers can achieve 1 part in 105 easily, timing with software with a typical operating system is more challenging!! After observing some anomalous results from my own tests under Windows 11 and Rust, I tried, so far unsuccessfully (see later), to achieve some more consistently. So, in this experiment, you should attempt to achieve some better results and report your observations (or complaints – if you also observe apparently incorrect results). We know from previous trials, that it is necessary to time a large number of operations (the nanosecond resolution apparently returned by Rust is clearly unreliable!). For accuracy, first you must remove **overheads**: even a short loop

**const N: u64 = 1e6; for j in 0 .. N { /\* some trivial calc \*/ }**

takes time to manage the loop. So you must subtract the loop overhead time. In the sample code, you will see a empty\_loop – a loop that does *almost* nothing:

**fn empty\_loop( n: u64 ) -> f64**

**First caution**: modern optimizing compilers are quite smart:

**fn t\_loop {  
 let mut x = 0; for j in 0 .. N { x += 1; }  
 }**

A good optimizing compiler will notice that the value calculated is not actually used, so decide to delete the code for the whole loop – it doesn’t actually calculate anything needed! So you will observe that my **empty\_loop** actually calculates a value and returns it. This forces the compiler to generate code for the calculation, and thus take some time to calculate it!

**Second caution**: As already noted, the operating system interrupts your program often for many irrelevant tasks, e.g. re-drawing the cursor, because you touched the mouse, checking the networkd for new Line messages and even useful things, e.g. automatically saving your files in case a flood shorts your power. So you must time a large number of iterations to average out the variations in these overheads. Probably, you will need 106 iterations before the interruptions average out and you see stable times.

**Exercise 1**

Add a loop to the template code, which varies **n** from some trivial value, e.g. 1, to a much larger value. Steps in an arithmetic series, 100, 200, 300, … may not achieve any useful result – unless you are prepared to wait for a week. Try a geometric series: 1, 2, 4, 8, … so that you reach a large value quickly – a series like {**10*n* | *n* = 0 .. *N*}** may be even more efficient. Some useful functions to print total times and average times per iteration are provided – use them or modify to improve your output.

Record the average time per iteration and report (in summary form) your results. Determine how many iterations are needed to reach a stable value – the average time per iteration becomes reasonably constant.

**Exercise 2**

Add a function which tests the performance of the library sin() function. Now, you should measure the sin function in a loop and **subtract** the time for the empty loop. Again, test for a number of iterations. You may see some odd results for small values of n. Use the value you obtained in Exercise 1 to guide your choice of how many iterations are needed to achieve useful results.

**Exercise 3**

In a previous exercise, you were asked to build a look up table (LUT). My expectation was that the LUT would be much faster than the library sin() function, because many terms of the expansion would be needed to compute sin() to the precision needed for an f64 result! I was surprised to see that the library sin() was very fast and matched the look up table approach – as least for high end CPU with floating point hardware. (Don’t expect the same result for a small microprocessor without floating point hardware – anyone with an Arduino at home, might confirm this 😊.) So I set out to find out why. See if you can find the explanation too! Build a function

**sin\_series(x:f64,n:u64)->f64**

that computes sin by evaluating the Taylor series:

A picture containing text, clock

Description automatically generated

Run your function for a small range of values of **x** and find out how many terms in the series are needed to produce an accurate result. In the lab sheet, report your accuracy for some value of **x** (you can use **x** = p/6 – since you know the result 😊) for several values of **n** (start with 1), where **n** is the number of terms in the series. Compare the time for your function with the library function.

Hint: To make an efficient implementation, look up Horner’s Rule on the web, or just observe that ***x*3 *= x* × *x2, x*5 *= x*3 × *x*2**, … and ***fact(n) = n* × *fact(n-1)* 😊.***However expect that the hackers that produced Rust’s library have a few additional tricks to use, so you should only get close to the library time.*

**Answer the questions on the attendance sheet, have a TA sign it off.**

**This lab will be marked from results on your attendance sheet. The TAs will confirm that your sin\_series** was OK**.**

**For victims of Rust: Note that check points have been added to the steps listed. If you have any doubts, it is STRONGLY RECOMMENDED that you check with one of the TAs each step BEFORE proceeding to the next. At that point, you should also tell the TA what you are going to do next.**

**IF you are confident: You may skip the checkpoints and fill them in at the end.**

**Website: kris.kmitl.ac.th/clinic/Courses/Rust/**

|  |  |  |  |
| --- | --- | --- | --- |
| **Attendance** | **01286120** | **Elementary Systems Programming** | **11 Oct 2022** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Name (Thai script\*)** |  | **Student ID**  **65011**   |  |  |  | | --- | --- | --- | |  |  |  | |
| **(Latin characters -  as you enrolled)** |  |
| **\****Please write clearly: practice for one farang who is trying to improve* **😉** | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Ex 1** | |  |  |  | | --- | --- | --- | | n | Total time (s) | Time/iteration (ns) | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | | TA |
|  | My time stabilized after  iterations or ……………….. seconds |  |
| **Ex 2** | |  |  |  | | --- | --- | --- | | Library sin | | | | n | Total time (s) | Time/iteration (ns) | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |
|  | My time stabilized after  iterations or ……………….. seconds |  |
| **Ex 3** | |  |  |  | | --- | --- | --- | | sin\_series | | | | **n** | **sin\_series (x, n)** | Time/iteration (ns) | |  |  |  | |  |  |  | |  |  |  | |  |  |  | |  |  |  | | **TA**  Reasonable **fn** code? |
|  | Comment on your result. Do you know why the library function is fast? |  |