RAMAIAH UNIVERSITY OF APPLIED SCIENCES

ASSIGNMENT

Course Code	CSC305A
Course Name	Programming language principles
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Shikhar singh
Reg. No	17ETCS002168
Semester/Year	05/2017
Course Leader/s	Chaitra S.

Declaration Sheet			
Student Name	Shikhar singh		
Reg. No	17ETCS002168		
Programme	B.Tech	Semester/Year	05/2017
Course Code	CSC305A		
Course Title	Programming language Principles		
Course Date		to	
Course Leader	Chaitra S.		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

Question No. 1

Solution to Question No. 1:

1.1 Implementation of the application using C programming:

C is an imperative and procedural programming general purpose language.

a. Imperative: a programming paradigm where computation is done in terms of statements that changes a program state i.e. programmer has a significant control over memory.
b. Procedural: a step by step method of coding where progression follows a procedural approach to solve a problem in particular.
   Also, every step is progressive as per the algorithm designed by the user.

For the given problem, the program is written so as to facilitate the performance measures to be done in other parts of the problem.

These changes are as follows:

1. The type of good or service is not mentioned as the program primarily deals with the price itself.
2. Tax for all the 5 rates is calculated for a given price instead.
3. Inputs such as price and quantity of the good or service is taken as random integers between ranges of 1-10,000 and 1-10 respectively instead of user inputs to rule out the possibility of delay of execution time due to user.

PROGRAM SCREENSHOTS:

The C program for the tax calculation application is as shown in figure 1.1.a and 1.1.b:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//#define DEBUGi(var) printf(#var" : %d\n", var)
//#define DEBUGf(_var) printf(#_var" : %f\n", _var)


float gst_calculator(float price,float gst[],int y){

    float nettotal = 0.0f;
    // DEBUGf(y);

    float subtotal;
    float x= price;

    // DEBUGf(x);

    float total = 0;


    for(int i =0; i<5 ; i++){
        subtotal = x*y;
        // DEBUGf(subtotal);
        nettotal += subtotal+(subtotal*(gst[i]));
        total += subtotal+(subtotal*(gst[i]));
        printf("\n%d\t | %d\t\t |  Rs.%.3f\t\t | Rs.%.3f\t\t ",(int)gst[i],y,subtotal,nettotal);
    }



    // DEBUGf(nettotal);
    // DEBUGf(total);

    return nettotal;
}
```

Figure 1.1.a: the sub-routine to calculate tax

in the figure 1.1.a, the source code of the subroutine for the calculation of the GST tax is shown.

It contains the required pre-processors for particular operations like generating a random number, generating the time seed for the random number generator etc. Below that are the general definitions of functions to debug the program (these are commented as there is no need for them now.).

The algorithm for the GST calculator sub routine is as follows:

1. Function definition header with required arguments i.e. price, gst[] (array for all the slabs) and y which is the quantity of goods.
2. Declare counters like subtotal (original price of the goods x quantity), net total(subtotal + subtotal x gst rate).
3. Run a loop to calculate the tax for each of the gst slabs in the array.
    a. Subtotal = quantity * original price
    b. Nettotal = subtotal + (subtotal * gst rate)
    c. Print the gst rate, quantity of the good , subtotal and the nettotal for the corresponding GST rate.
4. Return nettotal.

```c
int main(){

    // taking goods or service price randomly.
    srand(time(0));

    int price; int quantity;
    float net_total = 0.0 ;
    long int count = 100000;
    //assigning the gst slabs to calculate taxed price.
    float gst_slab[5] = { .0, .05, .12, .18, .28};
    printf("Program running ........");
    //printf("\nActual Price and Bill of Goods and Services as per the slabs :\n ");

    for(long int i =0 ; i<count; i++){

       //generating price randomly between range (1-10,000)
        price = (rand()%10000)+1;
       //generating quantity of the goods between range (1-10)
        quantity = (rand()%10)+1;

        //indenting proper format
        printf("\n\nPrice : %d",price);
        printf("\n\n------------------------------------------------------------------ ");
        printf("\nslab\t | quantity\t | sub_total\t\t\t  | net price  ");
        printf("\n------------------------------------------------------------------");

        // calling the tax calculator function
       net_total +=  gst_calculator(price,gst_slab,quantity);

    }

    //printing the total no. of items computed
    printf("\n total no. of unique items = %d",count);
    //printing the net cost for all the items.
    printf("\n total price = %f",net_total);
    printf("\nExiting the program.......");
    return 0;
}
```

Figure 1.1.b: driver function (main)

Figure 1.1.b: shows the driver function i.e. main which generates the inputs using random generators and calls the gst calculator for computation.

The random numbers are generated using in-built rand function which uses another function called srand(). The srand() function takes time(0) as a seed which allows non-deterministic and uniform distribution of the random numbers and prevent the same set of numbers from generating at each program execution.

The set of inputs are:

1. Price (between Rs. 1-10,000 in range.)
2. Quantity (between 1-10 units in range.)
3. GST slabs (all the rates in fractions stored in an array)

Then, the function is called with suitable arguments like price, gst_slab and quantity. The return value of the function is stored in the nettotal counter which is counting the total taxed price for all the goods in the program.

The data is tabulated and shown as output.

OUTPUT:

```
shikhar@sinewolf  ~/Documents/sem_5/assignments/plp  ./a.out
Program running ........
Actual Price and Bill of Goods and Services as per the slabs :


Price : 4830

---------------------------------------------------------------
slab     | quantity       | sub_total                | net price
----------------------------------------------------------------
0        | 6              |  Rs.28980.000            | Rs.28980.000
5        | 6              |  Rs.28980.000            | Rs.30429.000
12       | 6              |  Rs.28980.000            | Rs.32457.600
18       | 6              |  Rs.28980.000            | Rs.34196.398
28       | 6              |  Rs.28980.000            | Rs.37094.398

Price : 5276

---------------------------------------------------------------
slab     | quantity       | sub_total                | net price
----------------------------------------------------------------
0        | 2              |  Rs.10552.000            | Rs.10552.000
5        | 2              |  Rs.10552.000            | Rs.11079.600
12       | 2              |  Rs.10552.000            | Rs.11818.240
18       | 2              |  Rs.10552.000            | Rs.12451.360
28       | 2              |  Rs.10552.000            | Rs.13506.561

Price : 6403

---------------------------------------------------------------
slab     | quantity       | sub_total                | net price
----------------------------------------------------------------
0        | 9              |  Rs.57627.000            | Rs.57627.000
5        | 9              |  Rs.57627.000            | Rs.60508.352
12       | 9              |  Rs.57627.000            | Rs.64542.238
18       | 9              |  Rs.57627.000            | Rs.67999.859
28       | 9              |  Rs.57627.000            | Rs.73762.562
```

Figure 1.1.c:output

```
Price : 253

---------------------------------------------------------------
slab     | quantity       | sub_total                | net price
----------------------------------------------------------------
0        | 9              |  Rs.2277.000          | Rs.2277.000
5        | 9              |  Rs.2277.000          | Rs.2390.850
12       | 9              |  Rs.2277.000          | Rs.2550.240
18       | 9              |  Rs.2277.000          | Rs.2686.860
28       | 9              |  Rs.2277.000          | Rs.2914.560

Price : 9525

---------------------------------------------------------------
slab     | quantity       | sub_total                | net price
----------------------------------------------------------------
0        | 5              |  Rs.47625.000            | Rs.47625.000
5        | 5              |  Rs.47625.000            | Rs.50006.250
12       | 5              |  Rs.47625.000            | Rs.53340.000
18       | 5              |  Rs.47625.000            | Rs.56197.500
28       | 5              |  Rs.47625.000            | Rs.60960.000
Exiting the program.......
```

Figure 1.1.d: output

The entire data is tabulated and shown as output. Figure 1.1.c and Fig 1.1.d shows the table of the calculated tax for each gst slab and for each price of the goods and services.
The code for compilation and execution of the program is mentioned in Appendix A.

1.2 Implementation of the application using JAVA-programming using object-oriented features:
JAVA is an object-oriented programming (OOP) language. Which means that it is based upon objects that (having both data and methods) aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming are–

- Bottom–up approach in program design.
- Programs organized around objects, grouped in classes.
- Focus on data with methods to operate upon object's data.
- Interaction between objects through functions.
- Reusability (Inheritance) of design through creation of new classes by adding features to existing classes.

For the given problem, the program is written so as to facilitate the performance measures to be done in other parts of the problem. These changes are same as done in problem solution 1.1.

PROGRAM SCREENSHOTS:
The java program for the tax calculation application is as shown in figure 1.2.a and 1.2.b:

```java
import java.util.Random;

public class Tax_calculation {

    /**
     * @param args the command line arguments
     */
    public static double gst_calculator(double price,double gst[],int y){
       System.out.printf("\n\nPrice : %f",price);


       double nettotal=0.0f,subtotal;


       for(int i=0; i<5 ; i++)
       {

          //calculating the subtotal i.e. price of a good * quantity
           subtotal = price*y;
           //nettotal i.e. total price including gst tax.
           nettotal= subtotal+ (subtotal*gst[i]) ;
           //printing the final prices
           System.out.printf("\n%d\t | %d\t\t |  Rs.%.3f\t\t | Rs.%.3f\t\t ",(int)gst[i],y,subtotal,nettotal);

       }

       return nettotal;
    }
```

Figure 1.2.a: function for gst calculation

in the figure 1.2.a, the source code of the subroutine for the calculation of the GST tax is shown.
It contains the required libraries for particular operations like generating a random number.

The algorithm for the GST calculator method is as follows:

1. Method function definition header with required arguments i.e. price, gst[] (array for all the slabs) and y which is the quantity of goods.
2. Declare counters like subtotal (original price of the goods x quantity), net total(subtotal + subtotal x gst rate).
3. Run a loop to calculate the tax for each of the gst slabs in the array.
    a. Subtotal = quantity * original price
    b. Nettotal = subtotal + (subtotal * gst rate)
    c. Print the gst rate, quantity of the good,
    subtotal and the nettotal for the corresponding GST rate.
4. Return nettotal.

```java
public static void main(String[] args) {

    System.out.printf("\n");
    Random rand = new Random();

    double gstSlab[]= {0.0, 0.5, 0.12, 0.18, 0.28};
    int i=0, quantity;
    double price;
    int count= 100000;
    double net_total=0.0f;
    System.out.println("\nActual Price and Bill of Goods and Services as per the slabs :\\n ");

    for(i=0; i<count; i++)
    {
        //creating a random input for price (range 1 to 10,000)
        price = rand.nextInt(10000)+1;

        //creating a random input for quantity (range 1 to 10)
        quantity = rand.nextInt(10)+1;

        //printing everything in a tabular format
        System.out.printf("\n\nPrice : %f",price);
        System.out.printf("\n\n------------------------------------------------------------------ ");
        System.out.printf("\nslab\t | quantity\t | sub_total\t\t\t  | net price  ");
        System.out.printf("\n--------------------------------------------------------------------");


       net_total += gst_calculator(price,gstSlab,quantity);

    }


        //total no. of unique items
        System.out.printf("\ntotal no. of items : %d", count);
        //total taxed price for all the items
        System.out.printf("\ntotal price: %f",net_total);
}
```

Figure 1.1.b: driver function

The random numbers are generated using in-built rand(). The nextInt() class method is attached to this method to provide range of numbers and one to scale the range from 0 to 9,999 to 1 to 10,000.
The set of inputs are:

1. Price (between Rs. 1-10,000 in range.)
2. Quantity (between 1-10 units in range.)
3. GST slabs (all the rates in fractions stored in an array)

Then, the method is called with suitable arguments like price, gst_slab and quantity. The return value of the function is stored in the nettotal counter which is counting the total taxed price for all the goods in the program.

The data is tabulated and shown as output.

OUTPUT:

```
run:

Actual Price and Bill of Goods and Services as per the slabs :\n

Price : 2223.000000

---------------------------------------------------------------
slab     | quantity      | sub_total                       | net price
----------------------------------------------------------------

Price : 2223.000000
0        | 10            |  Rs.22230.000                   | Rs.22230.000
0        | 10            |  Rs.22230.000                   | Rs.33345.000
0        | 10            |  Rs.22230.000                   | Rs.24897.600
0        | 10            |  Rs.22230.000                   | Rs.26231.400
0        | 10            |  Rs.22230.000                   | Rs.28454.400

Price : 3449.000000

---------------------------------------------------------------
slab     | quantity      | sub_total                       | net price
----------------------------------------------------------------

Price : 3449.000000
0        | 1             |  Rs.3449.000            | Rs.3449.000
0        | 1             |  Rs.3449.000            | Rs.5173.500
0        | 1             |  Rs.3449.000            | Rs.3862.880
0        | 1             |  Rs.3449.000            | Rs.4069.820
0        | 1             |  Rs.3449.000            | Rs.4414.720

Price : 7989.000000

---------------------------------------------------------------
slab     | quantity      | sub_total                       | net price
----------------------------------------------------------------

Price : 7989.000000
0        | 6             |  Rs.47934.000                   | Rs.47934.000
0        | 6             |  Rs.47934.000                   | Rs.71901.000
0        | 6             |  Rs.47934.000                   | Rs.53686.080
0        | 6             |  Rs.47934.000                   | Rs.56562.120
0        | 6             |  Rs.47934.000                   | Rs.61355.520

Price : 2917.000000
```

Figure 1.2.c: output

```
---------------------------------------------------------------
slab     | quantity      | sub_total                    | net price
----------------------------------------------------------------

Price : 7989.000000
0        | 6             |  Rs.47934.000                | Rs.47934.000
0        | 6             |  Rs.47934.000                | Rs.71901.000
0        | 6             |  Rs.47934.000                | Rs.53686.080
0        | 6             |  Rs.47934.000                | Rs.56562.120
0        | 6             |  Rs.47934.000                | Rs.61355.520

Price : 2917.000000

---------------------------------------------------------------
slab     | quantity      | sub_total                    | net price
----------------------------------------------------------------

Price : 2917.000000
0        | 6             |  Rs.17502.000                | Rs.17502.000
0        | 6             |  Rs.17502.000                | Rs.26253.000
0        | 6             |  Rs.17502.000                | Rs.19602.240
0        | 6             |  Rs.17502.000                | Rs.20652.360
0        | 6             |  Rs.17502.000                | Rs.22402.560

Price : 2098.000000

---------------------------------------------------------------
slab     | quantity      | sub_total                    | net price
----------------------------------------------------------------

Price : 2098.000000
0        | 5             |  Rs.10490.000                | Rs.10490.000
0        | 5             |  Rs.10490.000                | Rs.15735.000
0        | 5             |  Rs.10490.000                | Rs.11748.800
0        | 5             |  Rs.10490.000                | Rs.12378.200
0        | 5             |  Rs.10490.000                | Rs.13427.200
```

Figure 1.2.d: output

The entire data is tabulated and shown as output. Figure 1.1.c and Fig 1.1.d shows the table of the calculated tax for each gst slab and for each price of the goods and services.

1.3 Discussion on the ease of writing program in C in comparison with that in JAVA:

The comparison on the writability criteria for an appropriate language depends entirely on its application. C is a low-level structural language. Thus, it is very primitive and less feature packed, which makes it easier to implement for general programming problems like this GST tax calculation problems. Whereas, JAVA is a pure OOP (Object Oriented Programming) based language and contains huge variety of complex features which increases the complexity of programming for general programming problems.

For e.g.: in figure 1.2.a, while defining the class method gst_calculator() in JAVA, we assign different attributes to methods like its visibility, type, scope i.e. public, static, double etc. which shoots the writability complexity of the program as compared to sub-routine in C (fig 1.1.a), which only has the type attribute.

Furthermore, C gives its users a significant control over the memory management of the system which makes it easier to update variable states. Whereas, JAVA being Object Oriented, contains fundamentals such as encapsulation and abstraction which prevents a direct change in states by restricting the access to them. They can only be accessed by using special methods like constructors, getters and setters. The JVM does the memory management and prevents the control of deleting variables by the programmer.

In C, the array declaration is easier than in JAVA but also makes it less secure. The arrays generated in C are not generic i.e. they cannot change their values during program runtime. Therefore, arrays have to be made generic manually by the user by constructing an Abstract Data Type (ADT). But that means all the functions associated with it are to be created by the programmer which is a tiresome process (needs a lot of effort) and increases the risks of memory leaks; but this also enables the programmer to gain full control over those associated functions and design them in any way he/she wants. Whereas in JAVA, there are certain methods which implicitly makes the arrays generic and contains predefined functions associated with them which cannot be modified.

Hence, it can be said that in C programming, the writability is simpler as compared to JAVA which makes it easier to program generic solutions, but it decreases the efficiency while programming enterprise applications, working with GUI, frameworks etc. Whereas, JAVA comes packed with libraries containing hundreds of associated methods which does the primitive work and lets the programmer work on higher complexities when working on high level programming problems.

1.4 Discussion on the amount of changes required to introduce a new tax slab 25% and removing 28% slab in both the languages.

```
double gstSlab[]= {0.0, 0.5, 0.12, 0.18, 0.28};
```

*Figure 1.4.a: GST slab array in **JAVA***

Fig. 1.4.a is a GST slab array declared in JAVA. It contains all the slab values in fractions simply to reduce computation. Also, it is dynamic and hence, new values can be added easily.

```
float gst_slab[5] = { .0, .05, .12, .18, .28};
```

*Figure 1.4.b: GST slab array in **C***

Fig. 1.4.b is a GST slab array declared in C. It contains all the slab values in fractions simply to reduce computation. Also, it is static and hence, new values can't be added directly. Either the entire array must be copied into a new array along with the new modification or the array must be made dynamic.

The implementation of GST slabs in this program in both the languages is done in a primitive way to facilitate the performance of the program. Generally, JAVA has a lot of in-built methods to deal with array modifications and JVM does the memory management which makes it very simple to induce new modifications. On the other hand, C is very primitive i.e. there is a very rare presence of language supported functions and memory management is done by the programmer itself due to which new modifications are difficult to do in C arrays. This leaves user to only two choices;

1. Make the array dynamic using malloc, calloc etc. in the first place.
2. Copy the entire array into a new array with the modifications.

As the GST slabs are predefined arrays in both the languages, thus there's only one change that had to be done i.e. replacing the 28% value in the array with 25% value in the array. (0.28 to 0.25.)

1.5 Discussion on the efficiency in terms of CPU and memory usage using tools in both the languages:

NOTE: The tools used in the performance and memory usage measurement are perf and memusage in linux. Their additional info is given in Appendix B.

In order to calculate the efficiency and memory usage of both the languages, we do some necessary changes which are as follows:

1. Suppressing all the unnecessary output streams to maximize CPU cycles by commenting them out. (fig 1.1.a, 1.1.b, 1.2.a, 1.2.b removing tabularized output.)
   The total number of unique goods or services along with the total taxed bill for all the goods with their quantity is printed instead.
2. Running the program for 100,000 and 10,000,000 cycles in order to shoot the CPU time to seconds from nanoseconds/microseconds for a valid quantity of comparison.
3. The programming methodology is kept as similar as possible in both the languages to keep comparison viable.

PERFORMANCE ANALYSIS:

In the figures 1.5.a and 1.5.b, are the performance analytics for C and JAVA done using perf tool in linux for 100,000 cycles.

```
Program running ........
 total no. of unique items = 100000
 total price = 15499666432.000000
Exiting the program.......
 Performance counter stats for './gst-c':

              8.07 msec task-clock:u              #    0.928 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
                55      page-faults:u             #    0.007 M/sec
        15,283,773      cycles:u                  #    1.895 GHz
        32,395,793      instructions:u            #    2.12  insn per cycle
         4,042,150      branches:u                #  501.082 M/sec
            10,537      branch-misses:u           #    0.26% of all branches

       0.008694892 seconds time elapsed

       0.005653000 seconds user
       0.002798000 seconds sys
```

Figure 1.5.a: performance measure in C using perf

```
Actual Price and Bill of Goods and Services as per the slabs :\n

total no. of items : 100000
total price: 3520433918.720002
 Performance counter stats for 'java -cp PLPAssignCorrect/bin/ plpassigncorrect.PLPAssignCorrect':

           225.70 msec task-clock:u              #    1.460 CPUs utilized
                0      context-switches:u        #    0.000 K/sec
                0      cpu-migrations:u          #    0.000 K/sec
            3,698      page-faults:u             #    0.016 M/sec
      384,772,984      cycles:u                  #    1.705 GHz
      400,470,245      instructions:u            #    1.04  insn per cycle
       75,408,645      branches:u                #  334.109 M/sec
        2,852,615      branch-misses:u           #    3.78% of all branches

      0.154600971 seconds time elapsed

      0.206589000 seconds user
      0.023726000 seconds sys
```

Figure 1.5.b: performance measure in JAVA using perf

From the following figures 1.5.a and 1.5.b, following things can be deduced:

1. The CPU utilization for C was 1.573 times less than that JAVA.
2. Execution time of C was 17.790 times faster than in JAVA.
3. CPU performance of C was 1.12 times better than JAVA.

Similarly, In the figures 1.5.c and 1.5.d, are the performance analytics for C and JAVA done using perf tool in linux for 10,000,000 cycles.

```
Program running ........
 total no. of unique items = 10000000
 total price = 1531339472896.000000
Exiting the program.......
 Performance counter stats for './gst-c':

           859.89 msec task-clock:u              #    0.998 CPUs utilized
                0      context-switches:u        #    0.000 K/sec
                0      cpu-migrations:u          #    0.000 K/sec
               55      page-faults:u             #    0.064 K/sec
    1,693,909,756      cycles:u                  #    1.970 GHz
    3,218,919,465      instructions:u            #    1.90  insn per cycle
      399,404,264      branches:u                #  464.484 M/sec
          670,369      branch-misses:u           #    0.17% of all branches

      0.861664718 seconds time elapsed

      0.843111000 seconds user
      0.009888000 seconds sys
```

*Figure 1.5.c: **C** program for 10,000,000 cycles*

```
Actual Price and Bill of Goods and Services as per the slabs :\n

total no. of items : 10000000
total price: 352032154050.549700
 Performance counter stats for 'java -cp PLPAssignCorrect/bin/ plpassigncorrect.PLPAssignCorrect':

            715.47 msec task-clock:u              #    1.183 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
             3,966      page-faults:u             #    0.006 M/sec
     1,327,361,507      cycles:u                  #    1.855 GHz
     1,354,003,653      instructions:u            #    1.02  insn per cycle
       165,123,209      branches:u                #  230.788 M/sec
         3,355,855      branch-misses:u           #    2.03% of all branches

       0.604682788 seconds time elapsed

       0.654885000 seconds user
       0.062190000 seconds sys
```

*Figure 1.5.d: **JAVA** program for 10,000,000 cycles*

From the following figures 1.5.c and 1.5.d, following things can be deduced:

1. The CPU utilization for C was 1.185 times less than that of JAVA.
2. Execution time of C was 1.424 times slower than in JAVA.
3. CPU performance of C was 1.06 times better than JAVA.

On an average of both the cycles:

1. The CPU utilization for C was 1.379 times less than that of JAVA.
2. Execution time of C was 9.607 times slower than in JAVA.
3. CPU performance of C was 1.09 times better than JAVA.

MEMORY USAGE:

The memory usage for 100,000 cycles for both C and JAVA programs is as follows:

```
Memory usage summary: heap total: 4096, heap peak: 4096, stack peak: 2144
         total calls   total memory   failed calls
 malloc|          1           4096              0
realloc|          0              0              0  (nomove:0, dec:0, free:0)
 calloc|          0              0              0
   free|          0              0
Histogram for block sizes:
 4096-4111            1 100% ==================================================
Program running ........
 total no. of unique items = 100000
 total price = 15499666432.000000
Exiting the program.......
```

Figure 1.5.e: memory usage in C using memusage

```
Actual Price and Bill of Goods and Services as per the slabs :\n

total no. of items : 100000
total price: 3523134603.519986
Memory usage summary: heap total: 14859065, heap peak: 12914071, stack peak: 30688
         total calls   total memory   failed calls
 malloc|      11379       14622739              0
realloc|         38           1120              0  (nomove:0, dec:0, free:0)
 calloc|        110         235206              0
   free|       4534        2019018
Histogram for block sizes:
    0-15            378   3% ===
   16-31           5366  46% ==================================================
   32-47           2266  19% =====================
   48-63           1219  10% ===========
   64-79            374   3% ===
   80-95            149   1% =
   96-111            57  <1%
  112-127           165   1% =
  128-143           108  <1% =
  144-159            21  <1%
  160-175            25  <1%
  176-191            63  <1%
  192-207            18  <1%
  208-223            10  <1%
  224-239            36  <1%
  240-255            54  <1%
  256-271            23  <1%
  272-287            53  <1%
  288-303            68  <1%
  304-319            28  <1%
  320-335            18  <1%
  336-351           128   1% =
  352-367             2  <1%
  368-383            62  <1%
  384-399             9  <1%
  400-415             2  <1%
  432-447            65  <1%
```

Figure 1.5.f: memory usage in Java using memusage

```
 3024-3039            1  <1%
 3296-3311            8  <1%
 3360-3375            2  <1%
 3728-3743            3  <1%
 3840-3855            1  <1%
 4000-4015            1  <1%
 4064-4079            1  <1%
 4096-4111          140   1%
 4128-4143            1  <1%
 5392-5407            1  <1%
 7456-7471            4  <1%
 7584-7599            1  <1%
 8048-8063            1  <1%
 8064-8079            4  <1%
 8192-8207           69  <1%
 8240-8255           13  <1%
 9104-9119            1  <1%
10208-10223           1  <1%
10560-10575           1  <1%
16384-16399           4  <1%
29824-29839           1  <1%
32736-32751          38  <1%
32768-32783           1  <1%
32816-32831           4  <1%
32880-32895           1  <1%
33440-33455           1  <1%
33472-33487           1  <1%
33936-33951           1  <1%
34304-34319           1  <1%
34416-34431           1  <1%
35776-35791           1  <1%
36160-36175           1  <1%
36496-36511           1  <1%
36816-36831           1  <1%
36976-36991           2  <1%
42176-42191           1  <1%
42656-42671           1  <1%
43840-43855           1  <1%
46544-46559           1  <1%
48736-48751           1  <1%
51248-51263           1  <1%
53712-53727           1  <1%
57616-57631           1  <1%
62960-62975           1  <1%
64496-64511           1  <1%
   large             16  <1%
```

Figure 1.5.g: memory usage in JAVA using memusage

The memory usage for 10,000,000 cycles for both C and JAVA programs is as follows:

```
Memory usage summary: heap total: 4096, heap peak: 4096, stack peak: 2144
         total calls   total memory   failed calls
 malloc|          1           4096              0
realloc|          0              0              0  (nomove:0, dec:0, free:0)
 calloc|          0              0              0
   free|          0              0
Histogram for block sizes:
 4096-4111            1 100% ==================================================
Program running ........
 total no. of unique items = 10000000
 total price = 1531361230848.000000
Exiting the program.......
```

Figure 1.5.h: memory usage for C for 10,000,000 cycles

```
Actual Price and Bill of Goods and Services as per the slabs :\n

total no. of items : 10000000
total price: 351902666534.358500
Memory usage summary: heap total: 17765272, heap peak: 13709783, stack peak: 26240
         total calls   total memory   failed calls
 malloc|      11458      17528946             0
realloc|         38          1120             0  (nomove:0, dec:0, free:0)
 calloc|        110        235206             0
   free|       4579       4236569
Histogram for block sizes:
    0-15            378   3% ===
   16-31           5373  46% ==================================================
   32-47           2267  19% =====================
   48-63           1219  10% ===========
   64-79            377   3% ===
   80-95            156   1% =
   96-111            55  <1%
  112-127           171   1% =
  128-143           116  <1% =
  144-159            20  <1%
  160-175            25  <1%
  176-191            64  <1%
  192-207            17  <1%
  208-223            10  <1%
  224-239            36  <1%
  240-255            54  <1%
  256-271            22  <1%
  272-287            53  <1%
  288-303            68  <1%
  304-319            28  <1%
  320-335            18  <1%
  336-351           128   1% =
  352-367             2  <1%
  368-383            62  <1%
  384-399             9  <1%
  400-415             2  <1%
  432-447            65  <1%
  464-479             2  <1%
  480-495           144   1% =
  496-511             2  <1%
  512-527             3  <1%
  528-543             1  <1%
  544-559             1  <1%
  560-575             6  <1%
```

*Figure 1.5.i: memory usage for **JAVA** 10,000,000 cycles*

From the fig. 1.5.e, 1.5.f, 1.5.g, 1.5.h, 1.5.i, the total of stack peak and malloc in JAVA is very large as compared to C (approximately 32 times). Which means that the total memory consumption of C is very less as compared to JAVA.

malloc is the main routine in Linux library i.e. libc which allocates memory for the program at runtime, both the programs make memory allocation during program execution to make the comparison viable.

The memory allocated by malloc and the freed memory are an important comparison as that shows the possible occurrence of memory leaks in the program.

The vast difference in memory allocation is due to large amount of libraries and their methodologies packed in JAVA. Memory dump allocation shows this difference where about 46% of memory is allocated in chunks of 16-31 bytes in JAVA this is because various others fields are packed in the libraries that are being used to support the functionalities.

CONCLUSION:

Conclusion of the analysis shows that JAVA has much more overhead as compared to C because many in-built methodologies and complex data structures are being interpreted along with the function file which shoots up the execution time and decreases the overall CPU performance of the program.

NOTE: detailed report of perf analysis is shown in APPENDIX A.

Java performance report using perf:

```
# To display the perf.data header info, please use --header/--header-only options.
#
#
# Total Lost Samples: 0
#
# Samples: 3K of event 'cycles:u'
# Event count (approx.): 1325829359
#
# Overhead  Command          Shared Object       Symbol
# ........  ...............  ..................  ....................................
#
    12.59%  java             [JIT] tid 4816      [.] 0x00007f116c42aa8d
     9.65%  java             [JIT] tid 4816      [.] 0x00007f116c42ab1a
     8.96%  java             [JIT] tid 4816      [.] 0x00007f116c42aaee
     2.68%  java             [JIT] tid 4816      [.] 0x00007f116c42aa41
     2.36%  java             [JIT] tid 4816      [.] 0x00007f116c42aaff
     2.34%  java             [JIT] tid 4816      [.] 0x00007f116c42aa4b
     1.37%  java             [JIT] tid 4816      [.] 0x00007f116c42aa3b
     0.83%  java             [JIT] tid 4816      [.] 0x00007f116c429bdf
     0.76%  java             [JIT] tid 4816      [.] 0x00007f116c42aabb
     0.72%  java             libjvm.so           [.] 0x0000000000d14785
     0.72%  java             [JIT] tid 4816      [.] 0x00007f116c42ab06
     0.72%  java             [JIT] tid 4816      [.] 0x00007f116c42aa97
     0.69%  java             [JIT] tid 4816      [.] 0x00007f116c42aa30
     0.69%  java             [JIT] tid 4816      [.] 0x00007f116c42ab17
     0.69%  java             [JIT] tid 4816      [.] 0x00007f116c42aaaf
     0.69%  java             [JIT] tid 4816      [.] 0x00007f116c42aa12
     0.65%  java             [JIT] tid 4816      [.] 0x00007f116c42aa68
     0.65%  java             [JIT] tid 4816      [.] 0x00007f116c42aadc
     0.64%  java             [JIT] tid 4816      [.] 0x00007f116c429b65
     0.58%  java             [JIT] tid 4816      [.] 0x00007f116c42aacc
     0.56%  java             [JIT] tid 4816      [.] 0x00007f116c428d76
     0.54%  java             [JIT] tid 4816      [.] 0x00007f116c429406
     0.54%  java             [JIT] tid 4816      [.] 0x00007f116c42aa52
     0.50%  java             [JIT] tid 4816      [.] 0x00007f116c429bc7
     0.47%  java             [JIT] tid 4816      [.] 0x00007f116c42ab6e
     0.46%  java             [JIT] tid 4816      [.] 0x00007f116c42aa93
     0.44%  java             [JIT] tid 4816      [.] 0x00007f116c42a4a9
     0.40%  java             libjvm.so           [.] 0x0000000000d147b7
     0.40%  java             [JIT] tid 4816      [.] 0x00007f116c42aa75
     0.36%  java             [JIT] tid 4816      [.] 0x00007f116c42aa7d
     0.36%  java             libjvm.so           [.] 0x0000000000d147b1
     0.33%  java             [JIT] tid 4816      [.] 0x00007f116c429c59
     0.33%  java             [JIT] tid 4816      [.] 0x00007f116c429c5d
     0.33%  java             libjvm.so           [.] 0x0000000000d147ae
     0.33%  java             [JIT] tid 4816      [.] 0x00007f116c429c53
     0.32%  java             [JIT] tid 4816      [.] 0x00007f116c42aa64
```

```
0.01%  Sweeper thread   libjvm.so            [.] 0x0000000000729110
0.01%  C1 CompilerThre  libjvm.so            [.] 0x0000000000bccd30
0.00%  C1 CompilerThre  libjvm.so            [.] 0x0000000000b7ab97
0.00%  G1 Refine#0      libpthread-2.30.so   [.] __pthread_mutex_lock
0.00%  java             libc-2.30.so         [.] __ctype_init
0.00%  java             libjvm.so            [.] 0x00000000008568d7
0.00%  java             libc-2.30.so         [.] realloc
0.00%  VM Periodic Tas  libpthread-2.30.so   [.] __pthread_mutex_unlock_usercnt
0.00%  G1 Main Marker   libpthread-2.30.so   [.] __condvar_dec_grefs
0.00%  G1 Young RemSet  libpthread-2.30.so   [.] __pthread_disable_asynccancel
0.00%  G1 Main Marker   libpthread-2.30.so   [.] pthread_cond_timedwait@@GLIBC_2.3.2
0.00%  GC Thread#0      libpthread-2.30.so   [.] __pthread_disable_asynccancel
0.00%  Sweeper thread   libpthread-2.30.so   [.] pthread_cond_timedwait@@GLIBC_2.3.2
0.00%  G1 Young RemSet  libpthread-2.30.so   [.] __condvar_release_lock
0.00%  VM Thread        [unknown]            [k] 0xffffffff9ea00b07
0.00%  G1 Young RemSet  libpthread-2.30.so   [.] pthread_cond_timedwait@@GLIBC_2.3.2
0.00%  C1 CompilerThre  libpthread-2.30.so   [.] pthread_cond_timedwait@@GLIBC_2.3.2
0.00%  C2 CompilerThre  libjvm.so            [.] 0x0000000000729110
0.00%  C2 CompilerThre  [unknown]            [k] 0xffffffff9ea00b07
0.00%  java             libjvm.so            [.] 0x0000000000bbb9f2
0.00%  GC Thread#0      [unknown]            [k] 0xffffffff9ea00b07
0.00%  G1 Young RemSet  [unknown]            [k] 0xffffffff9ea00b07
0.00%  Sweeper thread   [unknown]            [k] 0xffffffff9ea00b07
0.00%  G1 Refine#0      [unknown]            [k] 0xffffffff9ea00b07
0.00%  G1 Main Marker   [unknown]            [k] 0xffffffff9ea00b07
0.00%  VM Periodic Tas  [unknown]            [k] 0xffffffff9ea00b07
0.00%  G1 Young RemSet  [unknown]            [k] 0xffffffff9ea0015f
0.00%  C1 CompilerThre  [unknown]            [k] 0xffffffff9ea0015f
0.00%  G1 Main Marker   [unknown]            [k] 0xffffffff9ea0015f
0.00%  C2 CompilerThre  [unknown]            [k] 0xffffffff9ea0015f
0.00%  G1 Refine#0      [unknown]            [k] 0xffffffff9ea0015f
0.00%  GC Thread#0      [unknown]            [k] 0xffffffff9ea0015f
0.00%  Sweeper thread   [unknown]            [k] 0xffffffff9ea0015f
0.00%  VM Periodic Tas  [unknown]            [k] 0xffffffff9ea0015f
0.00%  VM Thread        [unknown]            [k] 0xffffffff9ea0015f
```

```
0.33%  java             [JIT] tid 4816      [.] 0x00007f116c429c53
0.32%  java             [JIT] tid 4816      [.] 0x00007f116c42aa64
0.29%  java             [JIT] tid 4816      [.] 0x00007f116c42ab20
0.25%  java             [JIT] tid 4816      [.] 0x00007f116c42aa79
0.20%  java             [JIT] tid 4816      [.] 0x00007f1164974e97
0.18%  C1 CompilerThre  ld-2.30.so          [.] __tls_get_addr
0.18%  java             [JIT] tid 4816      [.] 0x00007f116c42aaf2
0.18%  java             [JIT] tid 4816      [.] 0x00007f116c42ab63
0.16%  java             libjvm.so           [.] 0x00000000005b0658
0.15%  java             libjvm.so           [.] 0x00000000006627a7
0.15%  java             [JIT] tid 4816      [.] 0x00007f116c42a4f6
0.14%  java             [JIT] tid 4816      [.] 0x00007f116c4290c8
0.14%  java             [JIT] tid 4816      [.] 0x00007f116c429bce
0.14%  java             [JIT] tid 4816      [.] 0x00007f116c42aa8a
0.14%  java             [JIT] tid 4816      [.] 0x00007f11649746ff
0.14%  java             [JIT] tid 4816      [.] 0x00007f11649780fb
0.13%  java             libc-2.30.so        [.] __vfprintf_internal
0.13%  java             ld-2.30.so          [.] do_lookup_x
0.13%  java             [JIT] tid 4816      [.] 0x00007f1164960a71
0.13%  java             [JIT] tid 4816      [.] 0x00007f116496553b
0.13%  java             libjvm.so           [.] 0x0000000000b7a5fb
0.13%  java             libjvm.so           [.] 0x0000000000d147d1
0.12%  java             libc-2.30.so        [.] __memmove_avx_unaligned_erms
0.11%  java             [JIT] tid 4816      [.] 0x00007f116c429b6f
0.11%  java             [JIT] tid 4816      [.] 0x00007f116c42a49b
0.11%  java             [JIT] tid 4816      [.] 0x00007f116c429451
0.11%  java             [JIT] tid 4816      [.] 0x00007f116c42ab56
0.11%  java             [unknown]           [k] 0xffffffff9ea00b07
0.11%  java             [JIT] tid 4816      [.] 0x00007f116c429403
0.11%  java             [JIT] tid 4816      [.] 0x00007f116c42aa9b
0.10%  java             [JIT] tid 4816      [.] 0x00007f1164970a81
0.10%  java             libc-2.30.so        [.] _int_malloc
0.10%  java             [JIT] tid 4816      [.] 0x00007f11649748f8
0.10%  java             [JIT] tid 4816      [.] 0x00007f1164978594
0.10%  java             libjvm.so           [.] 0x0000000000bd6e80
0.10%  java             [vdso]              [.] 0x0000000000000983
0.10%  java             libjvm.so           [.] 0x0000000000d19afb
0.10%  java             libjvm.so           [.] 0x0000000000c69f5d
0.10%  C2 CompilerThre  ld-2.30.so          [.] __tls_get_addr
0.10%  java             libjvm.so           [.] 0x0000000000b3803e
0.10%  java             libc-2.30.so        [.] malloc
0.10%  java             [JIT] tid 4816      [.] 0x00007f11649654c0
0.10%  java             libjvm.so           [.] 0x0000000000d147dc
0.10%  C1 CompilerThre  [unknown]           [k] 0xffffffff9ea00b07
0.09%  C1 CompilerThre  [JIT] tid 4816      [.] 0x00007f1164957096
0.09%  java             libjvm.so           [.] 0x0000000000d14b69
```

perf Examples

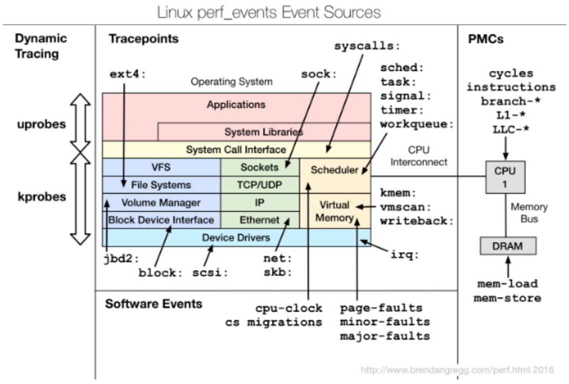

Image license: creative commons Attribution-ShareAlike 4.0.

These are some examples of using the perf Linux profiler, which has also been called Performance Counters for Linux (PCL), Linux perf events (LPE), or perf_events. Like Vince Weaver, I'll call it perf_events so that you can search on that term later. Searching for just "perf" finds sites on the police, petroleum, weed control, and a T-shirt. This is not an official perf page, for either perf_events or the T-shirt.

perf_events is an event-oriented observability tool, which can help you solve advanced performance and troubleshooting functions. Questions that can be answered include:

- Why is the kernel on-CPU so much? What code-paths?
- Which code-paths are causing CPU level 2 cache misses?
- Are the CPUs stalled on memory I/O?
- Which code-paths are allocating memory, and how much?
- What is triggering TCP retransmits?
- Is a certain kernel function being called, and how often?
- What reasons are threads leaving the CPU?

perf_events is part of the Linux kernel, under tools/perf. While it uses many Linux tracing features, some are not yet exposed via the perf command, and need to be used via the ftrace interface instead. My perf-tools collection (github) uses both perf_events and ftrace as needed.

Options controlling event selection

It is possible to measure one or more events per run of the perf tool. Events are designated using their symbolic names followed by optional unit masks and modifiers. Event names, unit masks, and modifiers are case insensitive.

By default, events are measured at **both** user and kernel levels:

```
perf stat -e cycles dd if=/dev/zero of=/dev/null count=100000
```

To measure only at the user level, it is necessary to pass a modifier:

```
perf stat -e cycles:u dd if=/dev/zero of=/dev/null count=100000
```

To measure both user and kernel (explicitly):

```
perf stat -e cycles:uk dd if=/dev/zero of=/dev/null count=100000
```

MEMUSAGE(1)   Linux user manual   MEMUSAGE(1)

NAME top

memusage - profile memory usage of a program

SYNOPSIS top

memusage [*option*]... *program* [*programoption*]...

DESCRIPTION top

memusage is a bash script which profiles memory usage of the program, *program*. It preloads the **libmemusage.so** library into the caller's environment (via the **LD_PRELOAD** environment variable; see ld.so(8)). The **libmemusage.so** library traces memory allocation by intercepting calls to malloc(3), calloc(3), free(3), and realloc(3); optionally, calls to mmap(2), mremap(2), and munmap(2) can also be intercepted.

memusage can output the collected data in textual form, or it can use memusagestat(1) (see the **-p** option, below) to create a PNG file containing graphical representation of the collected data.

Memory usage summary

The "Memory usage summary" line output by **memusage** contains three fields:

```
heap total
       Sum of size arguments of all malloc(3) calls, products of
       arguments (nmemb*size) of all calloc(3) calls, and sum of
       length arguments of all mmap(2) calls.  In the case of
       realloc(3) and mremap(2), if the new size of an allocation
       is larger than the previous size, the sum of all such
       differences (new size minus old size) is added.

heap peak
       Maximum of all size arguments of malloc(3), all products
       of nmemb*size of calloc(3), all size arguments of
       realloc(3), length arguments of mmap(2), and new_size
       arguments of mremap(2).

stack peak
       Before the first call to any monitored function, the stack
       pointer address (base stack pointer) is saved.  After each
       function call, the actual stack pointer address is read
       and the difference from the base stack pointer computed.
       The maximum of these differences is then the stack peak.
```

Bibliography

1. https://www.archlinux.org/packages/community/x86_64/perf/
2. https://wiki.archlinux.org/index.php/Improving_performance
3. https://perf.wiki.kernel.org/index.php/Tutorial
4. http://man7.org/linux/man-pages/man1/memusage.1.html