## Lab Session – Preliminaries and Profiling Tools

## Objectives.

* Learn how to compile and run C programs using Linux Terminal
* Write simple C programs using pointers
* Profile C/C++ programs using gprof profiler
* Profile C/C++ programs using perf tool
* Profile C/C++ programs using Valgrind tool

## Aim

The main **aim** of this lab session is to learn how to profile C/C++ applications. In this session, we will be using Linux as there are several well-known and well-used free tools such as gprof and Valgrind.

## Section1 – Preliminaries

## How to Compile and run a C/C++ program in Linux.

## In this lab session we will be using gedit text editor to write our programs and the Linux Terminal to compile and run our programs. Please note that there are more efficient programming environments to develop software such as Eclipse (it is free), but for small programs, this is not necessary.

## You can compile a .c file using the following command:

## *gcc source.c -o exec -other\_optional\_options*

## gcc is a well-known and used compiler. The executable name is specified just after the ‘-o’ (‘o’ stands for output). The rest options are optional.

## Then, you can run the executable by using the following command:

## *./exec*

Passing Input arguments – Argc, Argv

So far, all the programs we have written can be run with a single command. For example, if we compile an executable called exec, we can run it from within the same directory with the following command at the GNU/Linux command line:

*./exec*

However, what if you want to pass information from the command line to the program you are running?

Up until now, the skeletons we have used for our C programs have looked something like this:

*int main() { }*

From now on, our examples may look a bit more like this:

*int main (int argc, char \*argv[]) { }*

As you can see, main now has arguments. The name of the variable argc stands for "argument count"; argc contains the number of arguments passed to the program. The name of the variable argv stands for "argument vector". A vector is a one-dimensional array, and argv is a one-dimensional array of strings. Each string is one of the arguments that was passed to the program.

Compile the input\_arguments.c file using the following command gcc input\_arguments.c -o exec. Then run the program normally, but add some parameters too :

*./exec arg1 arg2*

The output will be

*argc = 3*

*arg[0] = "./p"*

*arg[1] = "arg1"*

*arg[2] = "arg2"*

**Task1. Study input\_arguments.c program**

Make sure you understand what this program does

## Using Pointers

**Using Pointers on 1-d arrays**

Every variable is stored into a memory location and every memory location has a memory address. A memory address can be accessed by using the ampersand (&) operator. Consider the following example

*#include <stdio.h>*

*int main () {*

*int var1=4;*

*printf("The memory address of %p contains %d \n", &var1, var1 );*

*return 0;*

*}*

The code above prints:

*The memory address of 0x7ffdd32f4b9c contains 4.*

Keep in mind that the memory address that var1 is stored might be different every time you compile the program. To print the memory address of a variable, the ‘%p’ format specifier is used. ‘%p’ is a format specifier which is used if we want to print data of type (void \*) or in simple words address of pointer or any other variable. The output is displayed in hexadecimal value. However, the memory addresses can be printed in integer format too, if we use ‘%d’ instead of ‘%p’ ; in this case, a warning will be shown ‘*warning: format ‘%d’ expects argument of type ‘int’, but argument 4 has type ‘int \*’, which can be ignored*.

**Task1. Study ‘pointers\_1d\_array\_initialization.c’ file.**

This file has a routine that initializes a one-dimensional array and three routines that print the array’s values, in three different ways, which are explained below. Please note that the array elements are always stored in consecutive memory locations (always).

1. printf("\n element %d equals to %d",i, A[i]);
2. printf("\n element %d equals to %d",i, \*(A+i) );
3. printf("\n element %d equals to %d",i, \*(ptr+i) );

The three bullets above are equivalent. What does ***\*(A+i)*** mean? A is an array, thus A is the memory address of the first array element. ***(A+i)*** is the memory address of the ith array element. \* refers to contents of memory address. Thus, ***\*(A+i)*** means : give me the content (value) of the ith array element. The same holds for the 3rd bullet as ***'ptr=&A[0]***', which means that the pointer shows to the memory address of A[0].

Compile and run the program using different ‘N’ values, to make sure that all the three routines print the right values.

In the C Programming Language, the ‘***#define***’ directive allows the definition of macros within your source code. This macro definition allows a constant value to be declared for use throughout your code. Macro definitions are not variables and cannot be changed by your program code like variables.

Keep in mind that in C language, all the routines must be declared before main function.

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. A pointer that is assigned NULL is called a null pointer. This is done by using

*int \*ptr = NULL;*

**Using Pointers on 2d arrays**

**Task2. Extend the program in ‘pointers\_1d\_array\_initialization.c’ file to 2-d arrays**

Make the one-dimensional array, two-dimensional and modify the four routines appropriately. Make sure your program prints the right values. Rename the new file as ‘pointers\_2d\_array\_initialization.c’. In C language, the array’s elements are always stored row-wise in memory; this means that first the elements of the first row are stored into memory, then the elements of the second row etc.

**Task3. Print the array’s memory addresses**

The ‘print\_array\_memory\_addresses.c’ file prints the memory addresses of the 2d array in the above example (pointers\_2d\_array\_initialization.c). The memory addresses are printed in hex format. This is further explained in task1 above.

How many bytes are allocated for every element? The answer is 4bytes, as the data type is of typr ‘int’ which is four bytes. As you can observe, the array’s elements are stored into consecutive memory locations. Change the data type of the array from ‘int’ to ‘short int’ and then into ‘long int’. Are the memory addresses different now? Why? Given that ‘short int’ occupies 2 bytes, each array element needs 2 bytes.

## Further Reading

If you want to learn more about C programming you can study the following links:

1. Tim Bailey, 2005, An Introduction to the C Programming Language and Software Design, available at: <http://www-personal.acfr.usyd.edu.au/tbailey/ctext/ctext.pdf>
2. C examples, Programiz, available at <https://www.programiz.com/c-programming/examples>
3. C Programming examples with Output, Beginners book, available at <https://beginnersbook.com/2015/02/simple-c-programs/>
4. C Programming Tutorial, from tutorialspoint.com, available at <https://www.unf.edu/~wkloster/2220/ppts/cprogramming_tutorial.pdf>

## Section 2 – Profiling Tools

## Gprof profiler

Gprof profiler is a tool which collects statistics on programs. It works by inserting appropriate code in the beginning and in the end of each function so as to collect information about the execution time. Gprof is not a debugger, so make sure your program is working. Gprof does not work for parallel programs.

Type the following command on terminal to see the manual of gprof

*man gprof*

**How to use gprof:**

**Step1:** Compile using ‘-pg’ option. This option adds extra code to the generated binary so as gprof can report detailed timing statistics.

**Step2:** Run the program normally

**Step3:** Type ‘gprof executable\_name -other\_optional\_options’

**Store the gprof results into a file**:

You can store the results into a file by adding the ‘>’ character. The complete command is:

*gprof executable\_name > results.txt*

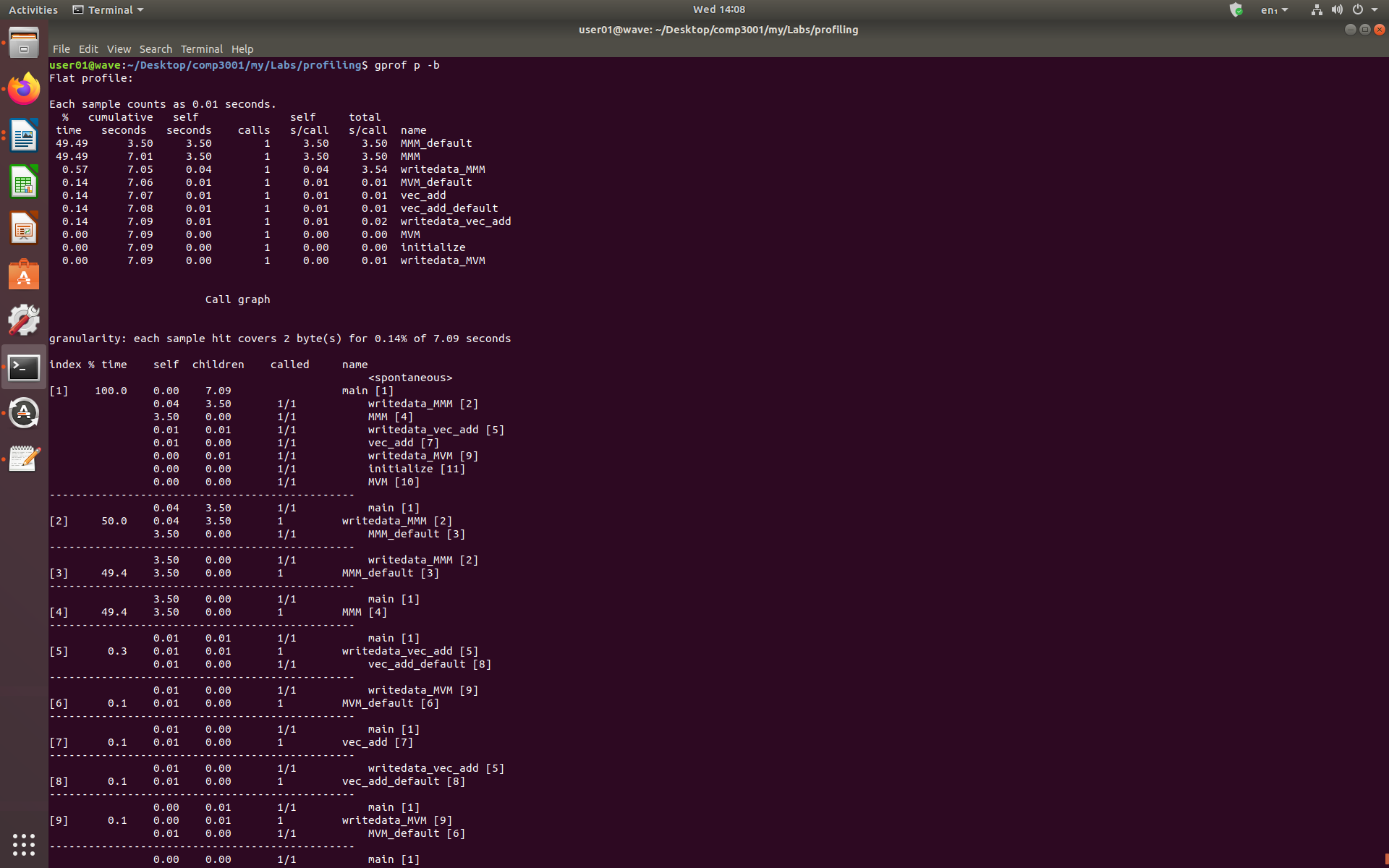
**Gprof options:**

Gprof profiler supports a large number of options. To find more about them use the man command as above. An helpful option is *‘-b’* that shows only the important information.

**Task1:** profile the benchmark.c file.

**Understanding Gprof output:**

The gprof output will look like Fig.1. The output is divided into two parts, i.e., Flat profile and Call graph.

*Fig.1 Gprof Output for benchmark.c program*

**Flat Profile**:

* The first column (%) shows the execution time percentage of each function.
* The second column (cumulative) shows the sum of the execution time in seconds by this function and those listed above it.
* The third column (self) shows the number of seconds accounted for by this function alone.
* The fourth column (calls) shows the number of times this function was invoked, if this function is profiled, else blank.
* The fifth column shows the average number of milliseconds spent in this ms/call function per call, if this function is profiled, else blank.
* The sixth column shows the average number of milliseconds spent in this ms/call function and its descendents per call, if this function is profiled, else blank.
* The name column shows the name of the function.

**Call Graph**: The call graph describes the call tree of the program. Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function.The lines above it list the functions that called this function, and the lines below it list the functions this one called.

* % time This is the percentage of the `total' time that was spent in this function and its children.
* self This is the total amount of time spent in this function.
* children This is the total amount of time propagated into this function by its children.
* called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.

## Further Reading

GPROF Tutorial – How to use Linux GNU GCC Profiling Tool, available at <https://www.thegeekstuff.com/2012/08/gprof-tutorial/>

## Valgrind Tool

The Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct. A detailed desciption is found in Valgrind’s webpage <https://www.valgrind.org/docs/manual/quick-start.html#quick-start.intro> . You can download Valgrind from <https://www.valgrind.org/downloads/?src=www.discoversdk.com> . You can install it by following the steps: a) extract the compressed file downloaded, b) use terminal and go to the valgrind directory and type: ‘.*/configure’*, then type ‘*make*’, then type ‘*make install*’.

In this session will be using Cachegrind tool of Valgrind. Cachegrind simulates how your program interacts with a machine's cache hierarchy. It simulates a machine with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). This exactly matches the configuration of many modern machines. For these CPUs with more than 2 levels of cache Cachegrind simulates the first-level and last-level caches only. Therefore, Cachegrind always refers to the I1, D1 and LL (last-level) caches. For more information read this <https://valgrind.org/docs/manual/cg-manual.html> .

**How to use Cachegrind**:

**Step1**: compile using ‘*-g*’ option

**Step2**: use the following command ‘*valgrind --tool=cachegrind ./executable*’

**Task1**: use valgrind to simulate how the benchmark.c file interacts with cache memories.

The output will look like this:

==11586== I refs: 1,041,351,336

==11586== I1 misses: 1,108

==11586== LLi misses: 1,101

==11586== I1 miss rate: 0.00%

==11586== LLi miss rate: 0.00%

==11586==

==11586== D refs: 387,398,347 (244,246,437 rd + 143,151,910 wr)

==11586== D1 misses: 411,589 ( 160,745 rd + 250,844 wr)

==11586== LLd misses: 405,386 ( 154,589 rd + 250,797 wr)

==11586== D1 miss rate: 0.1% ( 0.1% + 0.2% )

==11586== LLd miss rate: 0.1% ( 0.1% + 0.2% )

==11586==

==11586== LL refs: 412,697 ( 161,853 rd + 250,844 wr)

==11586== LL misses: 406,487 ( 155,690 rd + 250,797 wr)

==11586== LL miss rate: 0.0% ( 0.0% + 0.2% )

* ***Irefs***stands for Instruction references
* ***I1 misses***stands for L1 instruction cache misses
* ***LLi misses*** stands for Last level cache misses
* ***Drefs*** stands for data references
* ***D1*** misses stands for L1 data cache misses
* ***rd*** stands for reads while ***wr*** stands for writes

**Detailed Profiling**: Every time we run Valgrind, a file is stored into the working directory. This file name is ‘cachegrind.out.12065’, the last digits differ from simulation to simulation. To see a more detailed analysis use the following command:

*cg\_annotate cachegrind.out.12065*

For more annotate options see[*https://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-40/Nice/RuleRefinement/bin/valgrind-3.2.0/docs/html/cg-manual.html*](https://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-40/Nice/RuleRefinement/bin/valgrind-3.2.0/docs/html/cg-manual.html) *.*

## Further Reading:

1. The Valgrind Quick Start Guide, available at <https://www.valgrind.org/docs/manual/quick-start.html#quick-start.intro>

2. Cachegrind: a cache and branch-prediction, available at <https://valgrind.org/docs/manual/cg-manual.html>

## Perf Tool

The perf tool offers a rich set of commands to collect and analyze performance and trace data. A detailed description of perf can be found in <https://perf.wiki.kernel.org/index.php/Tutorial> .

**How to use Perf:** Perf includes many features. Some of them are listed below:

* *perf stat ./executable* : It shows CPU Performance statistics
* *perf stat -d ./executable* : It shows more detailed CPU Performance statistics
* *perf stat -d sleep 1 ./executable* : it shows detailed CPU Performance statistics, but by running the program only for 1 second
* *perf stat -d -C 0,2 ./executable* : it shows detailed CPU Performance statistics for the CPU core 0 and CPU core2.
* *perf stat -a ./p :*It shows CPU Performance statistics by using all the CPU cores
* *perf stat -d -e cycles ./executable* : shows detailed CPU Performance statistics including the number of CPU cycles by accessing the appopriate hardware counter.
* *perf stat -r 3 ./executable* : ‘r’ stand for repeat. -r 3, runs the program 3 times. It is possible to use perf stat to run the same program multiple times and get for each count, the standard deviation from the mean
* *perf list* : Lists all the options with regards to the hardware counters and events. The ‘-e’ option allows us to monitor the hardware counters. **You cannot display the values of the hardware counters by using a virtual machine.**
* *perf stat -d -C 0,1,2,3 -e cycles,instructions,ref-cycles,cpu-clock,cache-misses,cache-references,L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,L1-icache-load-misses,LLC-load-misses,LLC-loads,LLC-store-misses,LLC-stores,cache-misses,cache-references,mem-loads,mem-stores ./executable* : shows detailed statistics of the aforementioned hardware counters. Do not use space between two options
* perf stat -M Summary ./executable : in ‘perf list’ there are some sections such as Pipeline and Summary. To print those use -M option

**Perf record and report:**

* *perf* ***record*** *-e cycles,instructions,ref-cycles,cpu-clock,cache-misses,cache-references,L1-dcache-load-misses,L1-dcache-loads,L1-dcache-stores,L1-icache-load-misses,LLC-load-misses,LLC-loads,LLC-store-misses,LLC-stores,cache-misses,cache-references,mem-loads,mem-stores ./executable :* perf record collects samples which can then analyzed, possibly on another machine, using the perf report and perf annotate commands.
* perf report : By using the previous command (perf record), the samples collected are saved into a binary file called, by default, perf.data. The perf report command reads this file and generates a concise execution profile.

**Task1**: Use perf tool to profile the program in the benchmark.c file.

Keep in mind that the hardware counters are used by other processes too, not just by the program we are running.

## Further Reading

1. Tutorial, Linux kernel profiling with perf, available at <https://perf.wiki.kernel.org/index.php/Tutorial>

2. perf Examples, available at <http://www.brendangregg.com/perf.html>