

# **Micro-Benchmarking in Java, C and C#**

**Student:** Blaga Cristian-Ioan

**Group:** 30433

# Contents

1. Introduction.....	3
a. Requirements.....	3
b. Context.....	3
c. Objectives.....	3
2. Bibliographic Research.....	4
3. Analysis.....	5
4. Project Design.....	12
5. Implementation.....	16
6. Testing and validation.....	21
7. Conclusion.....	26
8. Bibliography.....	26

# 1. Introduction

## a. Requirements

Implementation of a bench-marking tool in 3 programming languages at choice in order to test execution time of different programs in these languages, with respect to:

- Memory allocation
- Memory deallocation
- Memory access time (both dynamic and static)
- Thread creation
- Thread context switch
- Thread migration

For each of the programming languages that have been chosen, we should see what can be done by using them, with respect to the operating system too.

## b. Context

The development of the project will be done on a VMware with ubuntu 16 as its operating system. The VM has the following specifications:

- 2 processors with 4 cores each
- 20GB HDD
- 2GB RAM

## c. Objectives

Week	Objective
Week 1 (February 28 <sup>th</sup> )	Choose semestrial project.
Week 2 (March 14 <sup>th</sup> )	Develop project outline, choose objectives and look for bibliographical resources.
Week 3 (March 28 <sup>th</sup> )	Java implementation of the benchmark (at least 50%).
Week 4 (April 11 <sup>th</sup> )	Java implementation finished, ask for opinions from TA, start C implementation.
Week 5 (April 25 <sup>th</sup> )	C implementantion at least 70%, ask for opinions from TA.
Week 6 (May 9 <sup>th</sup> )	C implementation finished, Python implementantion around 50%, ask TA for opinions.
Week 7(May 23 <sup>rd</sup> )	Project finished. Grading.

## 2. Bibliographical research

Below, it can be seen concretely what can we achieve in each programming language.

### i. Java

- In java, the most recommended way of measuring *elapsed time* is by use of the System.nanoTime() function.
- In order to get the most out of testing different features from the Java programming language, we will need to make use of some Java features that are as low level as possible. Thus, we will use the **sun.misc.Unsafe** class from Java for some of our tests.
- Memory allocation, in Java, is performed and managed automatically by the JVM. However, by use of the allocateMemory function from the Unsafe class we can achieve our desired behavior (together with the freeMemory, reallocateMemory, copyMemory, setMemory, etc. methods). [2],[3]
- Memory access measurement with help from getAddress, putAddress, putLong, etc.
- Measuring thread creation time in Java can be done by simply creating a Thread that does nothing and measuring the elapsed time between the moment before its creation and the moment it was created. [4]
- Thread context switch can be done by making use of the suspend and resume methods from the Thread class. [5]
- Thread migration can not be performed from Java (it will be performed using JNI).
- Of course, we will also make use of features from the JVM, like Runtime.getRuntime.exec().
- We will also try to make use of Java Native Interface, in order to write to be able to simulate a thread migration.

### ii. C

- In C, the elapsed time will be measured using the clock\_gettime function. [6]
- Memory allocation will be performed using malloc, calloc, realloc and free functions.
- Memory access time can be measured by allocating memory and reading a byte from the memory.
- Threads will be POSIX threads. That is, we will use Linux pthreads in our implementation. [7][8]
- Switching thread context can be performed, in Linux, only by us using locks and semaphores in order to concretely suspend and resume threads.

- Migrating threads can be done by making use of the sched\_setaffinity function.[9]

### iii. C#

- The elapsed time will be computed using the System.Diagnostics.Stopwatch class in order to be able to compute it as precise as possible.[13]
- Memory allocation can be performed by making use of the new operator and allocating an array of variable size.[12]
- Memory access time can be computed by randomly accessing a value from a previously allocated array.
- Thread creation can be measured by making use of the C# threads. They can be started with dummy functions in order to be able to concretely measure the creation time.
- Thread switch context can be performed by making use of the built-in thread functions Suspend and Resume.
- Thread migration can not be achieved in C#. However, as in Java, we will create a bridge to a C function that can achieve this.[14]

## 3. Analysis

As I have mentioned before, the project consists in developing a benchmarking application, written in three different programming languages, which will study the behavior of:

- Memory allocation
- Memory deallocation
- Memory access time
- Thread creation
- Thread context switch
- Thread migration

Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space. Memory allocation is the process of reserving a partial or complete portion of computer memory for the execution of programs and processes. Memory allocation is achieved through a process known as memory management.

Memory deallocation is done by the Operating System (OS) in order to free the Random Access Memory (RAM) of finished processes to be able to allocate new ones.

Memory access time is how long it takes for a character in memory to be transferred to or from the CPU.

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. A thread is strongly related to the Operating System it runs on.

A context switch is the process of storing the state of a process or of a thread, so that it can be restored and execution resumed from the same point later. This allows multiple processes to share a single CPU, and is an essential feature of a multitasking operating system.

Thread migration in operating system terminology actually means moving the thread from one core's run queue to another. This is done by the operating system scheduler to load balance the run queues of different cores available.

## Java

In java, we will implement the microbenchmark by making use of Java Native Interface. Since java runs on JVM, it is OS independent. We will need to implement some java native functions in order to access OS related functions.

Steps for using JNI[1]:

- a. Create a simple java class that uses a native method. Do not offer method implementation. Example:

*public static native Boolean sayHello(String name);*

- b. Inside this class, define a static block in which we will load the dynamic library that represents the interface between our class and the C implementation of the function. If we were to provide a name like "hello", the library that the system will search for is libhello.so.

```
static {  
    try {  
        System.loadLibrary("hello");  
    } catch(UnsatisfiedLinkError e) {  
        System.out.println(e);  
        System.exit(1);  
    }  
}
```

- c. In the command line, write:

*javac ClassName.java*

This will create the class file *ClassName.class*.

- d. In the command line, write:

*javah ClassName*

This will create the c header file *ClassName.h*.

- e. Offer a C implementation for the header file, *ClassName.c*.
- f. Run the command:

`g++ -I"$JAVA_HOME"/include -I"$JAVA_HOME"/include/linux -shared -fpic -o libhello.so ClassName.c`

This command will create a *libhello.so* dynamic library file to be used by our java class as an interface to the C code. JAVA\_HOME represents the path to the java installed location. In my case, this command is:

`g++ -I/usr/lib/jvm/default-java/include -I/usr/lib/jvm/default-java /include/linux -shared -fpic -o libhello.so ClassName.c`

g. Finally, run the following command:

`java -Djava.library.path=. ClassName`

This command runs the class file by specifying it where to find the dynamic library.

Thus, each feature will be tested as follows (most of the parameters for these measurements are chosen by the user and will be described in detail in a following section):

#### a. Elapsed time

Elapsed time will be measured by making two calls to the `System.nanoTime()` function: one before the functionality we want to measure, and one after the completion of that functionality. Since the function returns the number of nanoseconds since **1<sup>st</sup> January 1970**, a simple subtraction between the two values can precisely give us the elapsed time. This function will also measure the time in which the functionality **did not run** in situations like interrupts from the OS.

#### b. Memory allocation

Memory allocation will be performed by creating an integer array of a variable size (specified by the user) using the java operator **new**. By making use of this operator, we are sure that the memory is completely allocated before the function returns, thus allowing us to precisely measure the elapsed time.

#### c. Memory deallocation

In java, memory deallocation is done by the **Garbage Collector**. The garbage collector deallocates any object that does not have any explicit reference to it. Thus, when we set a reference variable to **null**, we are sure that at some point (not necessarily at that moment) the garbage collector will run. We could make an explicit call to it by making use of the `Runtime` class, but this will not ensure its execution in that moment. Nevertheless, we measure the time both with the explicit and implicit call to it and measure the time.

#### d. Memory access

In order to measure it more precisely and to be able to specify a variable size of my choice without introducing any metadata information from Java, I have decided to

make use of the Unsafe class for this feature. Thus, we will allocate a small array using the **allocateMemory()** method. Afterwards, randomly, we will access different bytes from this array using the **getByte()** method.

#### e. Thread creation

In java, threads are managed by the JVM. Thus, they are OS independent and may have functionalities that are not necessarily common for that specific OS. However, in java, threads can be created by making use of the Thread class, which needs as a constructor parameter a function that the thread will run. Afterwards, the thread can be started using the **start()** thread function. In order to wait for its execution to finish, we can make use of the **join()** function. Thus, we can simply implement thread creation time by passing to the thread class a dummy function that does nothing and waiting for the thread to finish its execution.

#### f. Thread switch context

In Linux, thread switch context can not be explicitly done by the user because the OS does not support it. However, java threads do not rely on Linux to run. They are managed by the JVM. Thus, the java allows us to manipulate the threads and perform switch contexts on them. These switches can be achieved by making use of the **suspend()** and **resume()** functions on threads that run indefinitely (infinite loop). After the behavior has been achieved, we can stop the infinite loop from the thread function by making use of a **volatile** boolean variable that can automatically stop it.

#### g. Thread migration

Thread migration is not supported by the JVM and can not be done in Java. However, in a Linux environment, it can easily be done in C. We will not go into details about how it is achieved in C (we will cover them in the C part). However, this function can be accessed by the java code by making use of the Java Native Interface. Basically, we will create a dynamic library (OS dependent) that allows us to execute C code from a Java code. The JNI details have been described above.

Each of the previously described measurements will be done a variable number of times. For each feature, a file will be populated containing, on line **ith**, the elapsed time for the **ith** try. This file will be saved in a special file structure with a concrete name that will be covered later.

After the execution of the microbenchmark was finalized, the program will call a python scrip with the intent of displaying the measured data.



## C

As I have previously stated, in C we will make use of **pthread**s. In order to be able to use them, the C files will need to be compiled using the **-lpthread** option as follows:

```
gcc -o c_benchmark C_Benchmark.c C_Logic.c -lpthread
```

The features will be implemented as follows:

### a. Elapsed time

The elapsed time will be measured by using the **clock\_gettime()** function. This function incorporates the same behavior as the one we used in the Java implementation. Thus, it will be used exactly the same. We also provide to it a parameter, **CLOCK\_MONOTONIC**, in order to be able to measure the time even if the OS interrupted the execution.[10]

### b. Memory allocation

Memory allocation will be implemented by making use of both the **calloc** and **malloc** functions. The sole difference between the two is the fact that the first one initializes the allocated memory area with 0. They both return a pointer to the beginning of the memory zone. The allocated size is variable.

### c. Memory deallocation

Memory deallocation will be implemented by making use of the **free** function which, given a pointer representing the beginning of a memory zone, it will deallocate it instantly when it is called.

### d. Memory access

Memory access will be implemented in the same manner as we have done it in Java. We will allocate a small array and, randomly, we will access an element from it. We will use only pointers for this part.

### e. Thread creation

The same logic will be applied here as in the Java implementation case. We will create a thread by making use of the **pthread\_create** function to which we will provide a dummy function for the thread to run. Afterwards, the thread will be started automatically. We will wait for it to finish its execution using **pthread\_join**.

### f. Thread switch context

Unfortunately, Linux does not allow the C programmer to manually suspend and resume threads at his will. Thus, we will need to manually implement these in order to simulate that behavior. In order to do this, I have made use of both **condition variables** and

**locks** in order to be able to concretely and precisely simulate the behavior. Basically, we will allow the thread to start and we will manually stop it from the main thread. The moment the execution passes from the thread to the main thread (which was blocked up until now), we have simulated **suspend**. Afterwards, we will allow it to run again. The moment it starts (the execution returns to the thread), we have simulated the **resume** behavior.

### g. Thread migration

This behavior, even if it is allowed by Linux, is harder to implement. Firstly, we need to find out on what core of the cpu the thread is running. This can be done by making use of the **sched\_getcpu** function. Afterwards, we need to see what cores are available for us to move the thread to. We can see these cores by calling the **CPU\_ISSET** function. The next step is to choose a core, randomly, that is different than the one on which our thread is running, to be the one on which we move our thread to. The final step is to actually move the thread by calling the **sched\_set\_affinity** function. (Only this step is measured).

Each of the previously described measurements will be done a variable number of times. For each feature, a file will be populated containing, on line **ith**, the elapsed time for the **ith** try. This file will be saved in a special file structure with a concrete name that will be covered later.

After the execution of the microbenchmark was finalized, the program will call a python scrip with the intent of displaying the measured data.

## C#

For the C# implementation, I have chosen to use the **Mono** compiler in order to be able to run C# on Linux. Thus, the command with which we compile the C# files is the following:

```
mcs -out:c#_benchmark c#_benchmark.cs c#_benchmark_logic.cs
```

As in the Java case, we can not test thread migration directly from C#. Thus, we will make a bridge to the previously defined function the following way:

- Move the function to another file; call it **c\_helper.c**
- Run the following command in the terminal in order to create a dynamic shared library:

```
gcc -shared -o libChelper.so -fPIC c_helper.c -lpthread
```

- In C# implementation, use this library by creating a function with no body that has an **extern** tag attached to it and annotate it with:

```
[DllImport("Chelper.so", EntryPoint="threadMigration")]
```

The rest of the features will be implemented as follows:

**a. Elapsed time**

The elapsed time will be computed by making use of the `StopWatch` class implemented in C#. We will use it exactly the same way we have used the other time measurement tools in the other two languages. However, the time is computed in **elapsed ticks**. In order to concretely measure the time, we need to convert the ticks (based on the frequency), to nanoseconds.

**b. Memory allocation**

Memory allocation is performed in the same way we have performed the memory allocation in java. We will allocate an array of variable size using the **new** operator.

**c. Memory deallocation**

As in java, C# has a built-in garbage collector that deallocates any objects that have no reference to them anymore. We can achieve this by allocating an object and then setting the reference to **null**. However, unlike in Java, the garbage collector can be explicitly called (and it will automatically run) by making use of the **System** class. We will measure the deallocation time with both the garbage collector on and off.

**d. Memory access**

Memory access will be implemented in the same manner as we have done it in Java. We will allocate a small array and, randomly, we will access an element from it.

**e. Thread creation**

Thread creation will be measured the same way we have done for the other two languages. We will create a new thread by making use of **Thread** class which needs a function sent to the constructor for the thread to run on. We will send a dummy function that does nothing. Afterwards, the thread can be started by calling the **Start()** function. We can wait for it to finish its execution by calling the **Join()** function.

**f. Thread switch context**

Exactly as Java, C# code does not directly run on the OS, but it runs on a virtual machine. Thus, we can easily test the switch context behavior by creating a thread that has an infinite loop and call on it **Suspend()** and **Resume()**. After we have finished testing the behavior, we can stop the threads by making use of a **volatile** variable that stops the infinite loop instantly.

**g. Thread migration**

Thread migration has been achieved by creating a bridge to the previously defined C function for this measurement. It can not be achieved in C# directly.

Each of the previously described measurements will be done a variable number of times. For each feature, a file will be populated containing, on line **ith**, the elapsed time for the **ith** try. This file will be saved in a special file structure with a concrete name that will be covered later.

After the execution of the microbenchmark was finalized, the program will call a python scrip with the intent of displaying the measured data.

## Data Filtering

For all the three languages, we have chosen to measure the elapsed time by also including the times in which the respective measurements were interrupted by the OS. Because we do this, we will introduce in our measurement the OS delay time. This is something we want because we want to precisely see the elapsed time taken for the certain measurement to take place. However, we will introduce some unwanted spikes in our data that, when they will be plotted (this will be covered in the next section), will make the more relevant data (the average) to not be seen. Thus, I have decided to ignore the maximum 10% of the data (in order to display only the relevant one and not the huge spikes from the OS that are not so common).

## 4. Project Design

The project is structured into five main components:

- Java User Interface
- C Microbenchmark
- Java Microbenchmark
- C# Microbenchmark
- Python plots

The three microbenchmarks we have described in the previous section can be called using their executable which receive six parameters:

- Memory allocation tries
- Memory allocation size
- Memory access tries
- Thread creation tries
- Thread switch context tries
- Thread migration tries

After the execution of one of these component is done, they save the measured data into a special file structure in the python plot component (in a folder with the name of the language of the microbenchmark which contains multiple folders with the name of the feature being tested; in these folders, we save multiple files containing different measurements). The last step of the execution of these components is to call the python plot component to plot the data.

## Python plot

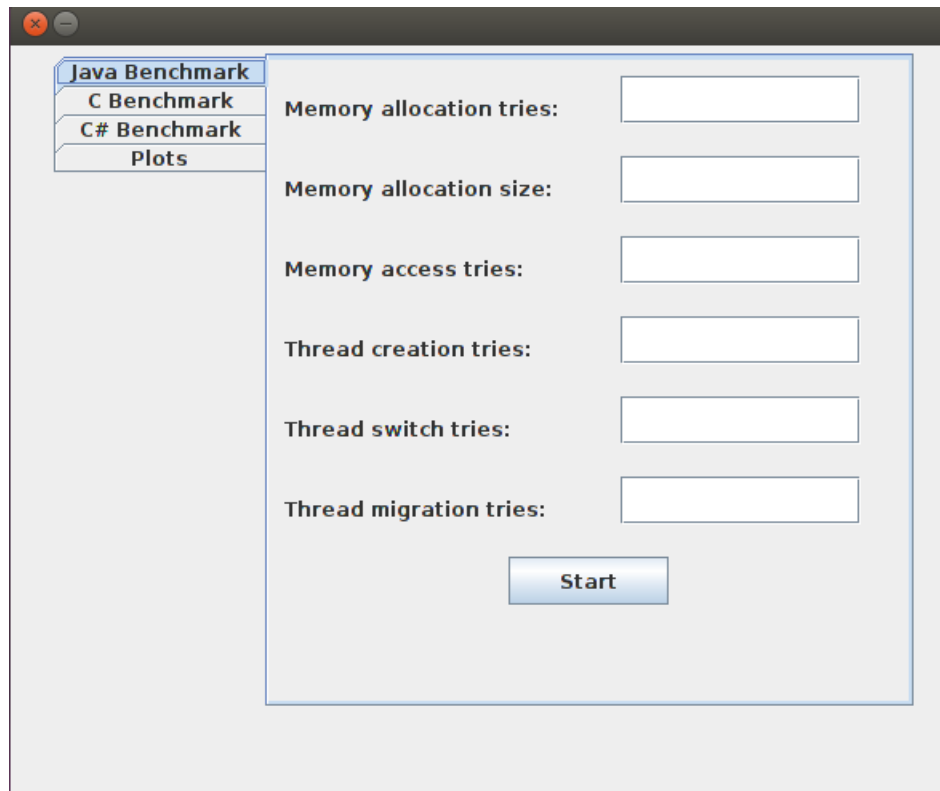
The python plot component is represented by a python script which can be configured by sending different command line parameters. Each microbenchmark calls it by specifying the language name. The python plot, based on the command line parameters, decides what data we want it to represent and reads the specific data. Afterwards, it filters the data as we have described in the previous section. Next, based on the parameters it has received, it will create a plot with one or more subplots. (screenshots will be included in the experiment/test section). Each subplot has the axis labeled and each line in the plot has a color. Each subplot contains a legend explaining what the lines represent.

The command line arguments the python plot can receive can be seen:

- java
- c
- c#
- all
- mem\_access MemoryAccessTime
- mem\_allocation MemoryAllocationTime
- mem\_deallocation MemoryDeallocationTime
- thread\_creation ThreadCreationTime
- thread\_switch\_context ThreadSwitchContextTime
- thread\_migration ThreadMigrationTime

## User Interface

The User Interface is implemented in **java swing**. It represents the main entry point to the application. From it, an user can basically do and manipulate everything the project has to offer. Firstly, for each language, it can run measurements by specifying the six parameters we have previously described for each microbenchmark.

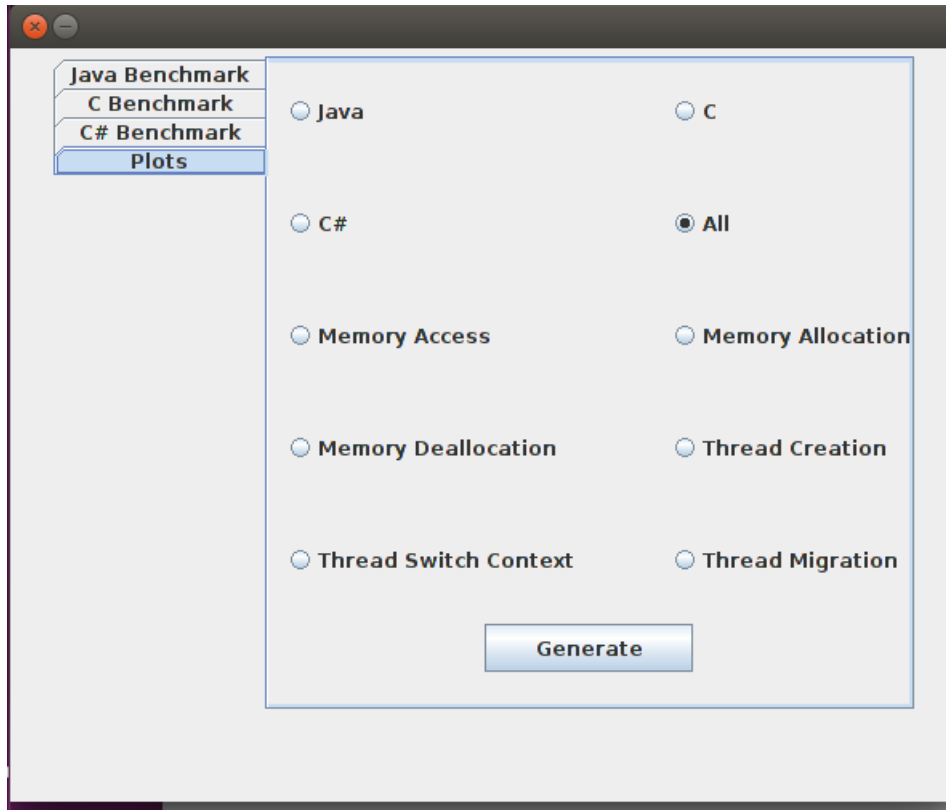


The screenshot shows a software window titled "Java Benchmark". On the left side, there is a vertical tab bar with four tabs: "Java Benchmark" (which is selected and highlighted in blue), "C Benchmark", "C# Benchmark", and "Plots". The main area of the window is divided into two sections. The top section contains six rows of labels followed by empty text input fields: "Memory allocation tries:", "Memory allocation size:", "Memory access tries:", "Thread creation tries:", "Thread switch tries:", and "Thread migration tries:". The bottom section contains a single blue button labeled "Start".

As it can be seen, on the left side, we have four tabs. The first three represent the microbenchmark we have previously described. For these three tabs, on the right side, we can specify the parameters we want to perform the measurements with. Afterwards, we can press the **start** button in order to start the measurement.

In this section, we also have some data validation on the User Interface. An user can only introduce numerical values that are strictly greater than 0. In case the user introduces a non-valid value (or did not introduce anything in a field), a pop-up appears specifying to the user how he should operate the User Interface.

Moreover, on the fourth tab, after the user has performed at least one measurement with each language, the user can choose to directly create the plots he wants to see (rather than waiting for the measurements to be done) by using directly the data previously computed. Moreover, from this section, the user can also combine different measurements for the same feature from different microbenchmarks under the same plot.



The buttons have the following functionalities:

**a. Java**

Create a plot containing six subplots with all the features tested using the Java microbenchmark.

**b. C**

Create a plot containing six subplots with all the features tested using the C microbenchmark.

**c. C#**

Create a plot containing six subplots with all the features tested using the C# microbenchmark.

**d. All**

Create a plot containing six subplots with all the features tested using all the microbenchmarks.

**e. Memory Access**

Create a plot containing only one subplot with the memory access measurement from all the three microbenchmarks.

**f. Memory Deallocation**

Create a plot containing only one subplot with the memory deallocation measurement from all the three microbenchmarks.

**g. Memory Allocation**

Create a plot containing only one subplot with the memory allocation measurement from all the three microbenchmarks.

**h. Thread Creation**

Create a plot containing only one subplot with the thread creation measurement from all the three microbenchmarks.

**i. Thread Switch Context**

Create a plot containing only one subplot with the thread switch context measurement from all the three microbenchmarks.

**j. Thread migration**

Create a plot containing only one subplot with the thread migration measurement from all the three microbenchmarks.

The application can simply be started by starting the User Interface of the application by running the following command:

```
java UserInterface
```

## **5. Implementation**

As I have previously said, this project is composed out of five components. In this section, I will go over each of them to discuss implementation details.

**a. User Interface**

The user interface is the main entry point to the application. It represents the gateway for the user to start benchmarks or visualize the different measurements he did. It is written purely in Java by making use of the java swing packages. This component is composed of the following files:

- MessageFrame.java

This class represents a pop-up window that is used when the user does not introduce any values into the UI or he inserts invalid values.



- User Interface.java

This class incorporates the entire User Interface window that was presented with screenshots in the previous section. It contains four tabs, three for the benchmarks in Java, C and C#, plus one tab for the user to be able to generate plots. I have made use of features from the Java Swing in this component like Jbuttons, JTabbedPane etc. in order to implement the component. It also features input validation for the microbenchmark part when the start button is pressed. Depending on the tab the user is on, the button calls different executables in order to achieve the desired behavior.

In order to be able to run this component, we need to first compile the two files by running the following commands:

```
javac MessageFrame.java
```

```
javac UserInterface.java
```

After running these two commands and creating the **.class** files, we can start the component by running the following command:

```
java UserInterface
```

## **b. Java Microbenchmark**

This component represents the implementation of the microbenchmark for testing the six features presented before in Java. Five of the features have been implemented in pure Java but one feature (thread migration) has been implemented in C and it is accessed by the Java microbenchmark by making use of the Java Native Interface. Thus, this component is composed of the following files:

- JavaBenchmark.java

This class incorporates the main entry point of the microbenchmark. It features the main function from which the component reads the parameters sent by the user to the command line. Afterwards, it starts, one by one, each measurement. There is one function for each measurement plus one function that prints the measurement to a file. As described in the previous section, each measurement needs to be saved in a file in a special file structure in order to be easily accessed by the python script when plotting the data. One special thing regarding this class is that it incorporates a **native** method, a method that represents the entry point to the measurement performed in C.

- **JavaBenchmark.h**

Represents the header file for the C implementation generated by the JNI.  
It represents the signature of the native method from the previous file.

- **JavaBenchmarkNativeInterface.c**

This file represents the concrete implementation of the previous header.  
This C function is the one called by the java implementation when it wants to access this functionality. This function will “redirect” this call to the actual C logic.

- **JavaBenchmarkNativeLogic.h**

This file represents the header for the C measurement of the thread migration feature. It represents the access point to the logic from the previous described file.

- **JavaBenchmarkNativeLogic.c**

This file represents the concrete implementation of the thread migration feature in the C language. It returns the elapsed time for the migration to happen.

- **libjavaBenchmark.so**

This file represents the dynamic library that the java benchmark uses in order to be able to access the C implementation.

In order to be able to run this component, several steps must be taken. First and foremost, we need to compile all the files whilst also making use of the JNI to create the bridge between Java and C. Moreover, we also need to create the dynamic library. Thus, the commands are:

```
javac JavaBenchmark.java
```

```
javah JavaBenchmark
```

```
g++ -I/usr/lib/jvm/java-8-openjdk-i386/include -I/usr/lib/jvm/java-8-openjdk-i386/include/linux  
-shared -fPIC -o libjavaBenchmark.so JavaBenchmarkNativeInterface.c  
JavaBenchmarkNativeLogic.c -lpthread
```

After the previous commands have been run, we can start the component as follows:

```
java -Djava.library.path=. JavaBenchmark <inline_arguments>
```

### c. C MicroBenchmark

This component represents the microbenchmark implemented purely in the C language. It contains all the six features that we test. The program can be run by calling the executable with the six configurable parameters. After each feature is tested, the component saves the measurements in a special file structure for the python plotter to display the data. Once the execution is done, it calls the python plotter. The component is made up of the following files:

- C\_Benchmark.c

This file represents the entry point of the user to this component. It contains the main function which receives the six configurable parameters the user has sent. Moreover, it contains functions that test each feature for the specified number of times. However, the features have not been implemented in this file. Moreover, it contains a function that prints all the measurements to the special file structure for the python plotter.

- C\_Logic.h

This file represents the header file for the actual implementation of each feature. Each feature is represented with its signature in this file. For each feature, by calling a function, the test is done only once. It is the job of the previous file to make the measurements for a variable number of tries.

- C\_Logic.c

This file represents the concrete implementation for the features being tested. It contains the implementation of all the functions present in the header file plus some helper or thread functions. All the functions present in the header file return the elapsed time for the specific feature that is being tested.

In order to use this component, the first step is to compile it by running the following command:

```
gcc -o c_benchmark C_Benchmark.c C_Logic.c -lpthread
```

Afterwards, the component can be started by running the following command:

```
./c_benchmark
```

### d. C# MicroBenchmark

This component represents the microbenchmark implementation for the six features we want to test in this application written in C#. However, the implementation is not purely done in C#. The thread migration measurement can not be performed in C#. Thus, as in the Java case, I

have created a bridge to a C implementation that tests this feature. Moreover, the executable can be run by sending to the component the six configurable parameters specified above. The component is made up of the following files:

- **c#\_benchmark.cs**

This file represents the entry point for the user to this component. It consists of the main function that takes receives the six configurable parameters sent by the user. Afterwards, it performs each measurement with respect to the parameters received. There is one function for each feature being measured. However, the concrete implementation of the tested features is not present here. All the functions from this file are solely responsible for coordinating the measurements, not for actually performing them. Moreover, after each measurement has been performed, the data is saved into a special file structure in the python plotter component for it to be able to display the data. When the execution of this component is finished, the python plotter is automatically called. One thing to note is that in this file we have an **extern** function which represents the gateway to the C implementation of the thread migration measurement.

- **c#\_benchmark\_logic.cs**

This file represents the concrete implementation for the five features that can be tested in C#. For each feature, only one measurement is performed. The orchestration of the measurement is performed by the previous file. Besides the methods that concretely test the features, there are also helper and thread methods in this file.

- **c\_helper.c**

This file represents the C implementation for the thread migration measurement that can not be tested from C#. It is accessed from the main file of this component through a bridge by using a dynamically shared library. The corresponding function from the main file through which we access this implementation is an **extern** function.

In order to be able to run this component, we first need to create the shared library by running the following command:

```
gcc -shared -o libChelper.so -fPIC c_helper.c -lpthread
```

Afterwards, the component can be run by performing the following command:

```
mcs -out:c#_benchmark c#_benchmark.cs/c#_benchmark.exe <inline_arguments>
```

#### **e. Python Plotter**

This component consists of a single python file and the special file structure we have talked in the previous sections. The python file, based on the arguments it receives when is run, selects only some of the files to be read. Afterwards, it filters the data by ignoring the 10% maximum values. Finally, it plots the data in different subplots for each feature by labeling both the plot and the lines in the plot. When it is done, it displays the plots.

As it can be seen, I have chosen to implement the project by modularizing all the components as much as possible. This decision was made in order to maximize the scalability and reusability of this project. Moreover, all the components are independent from each other (they basically communicate through files which, at any moment, can be changed). Thus, this project incorporates the idea of loosely couplings.

## **6. Testing and validation**

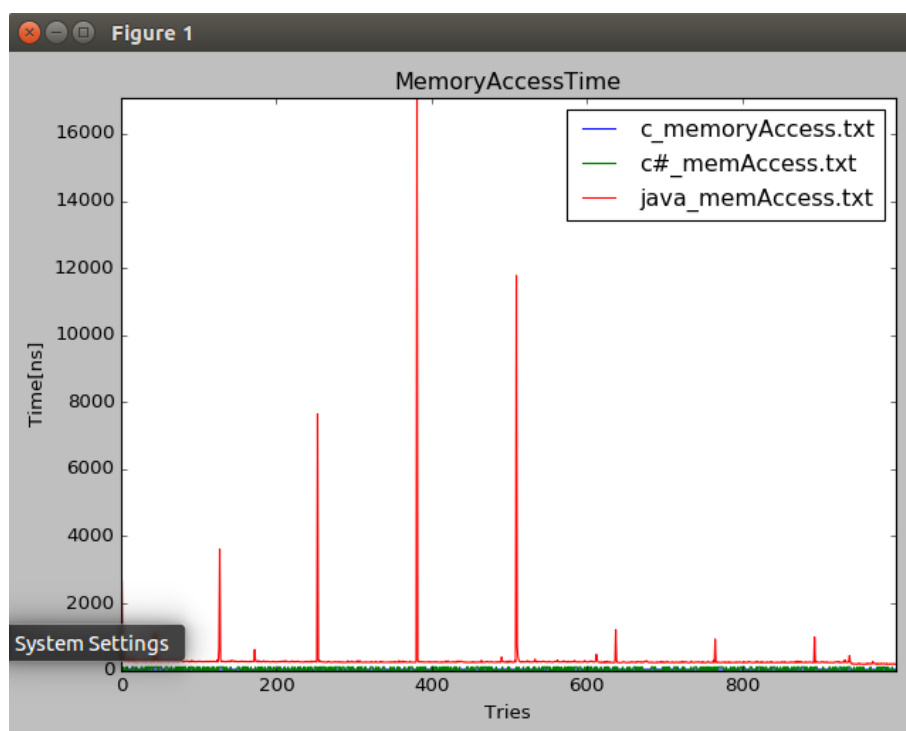
The experiments have been performed by running all the microbenchmarks (Java, C, C#) with the following configuration parameters:

- Memory allocation tries = 1000
- Memory allocation size = 1000000
- Memory access tries = 1000
- Thread creation tries = 1000
- Thread switch tries = 1000
- Thread migration tries = 1000

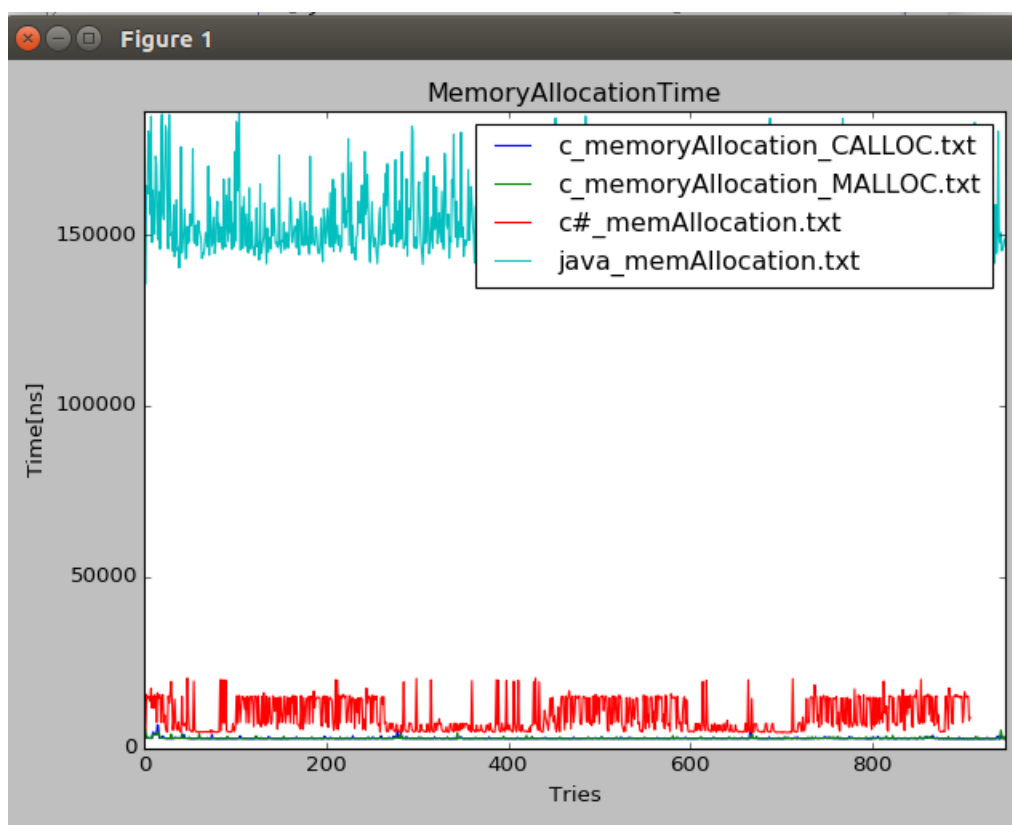
After all the microbenchmarks have run, we will now look at the plots individually for each feature in order to be able to see the results better.

#### **a. Memory Access**

As it can be seen from the generated plot, java memory access has some huge spikes comparative to all the other languages. However, if we ignore these spikes, it can easily be seen that java has a greater memory access time than both C and C# implementations. Moreover, the fastest one can be seen that is the C implementation, which is even more stable than the C# one.

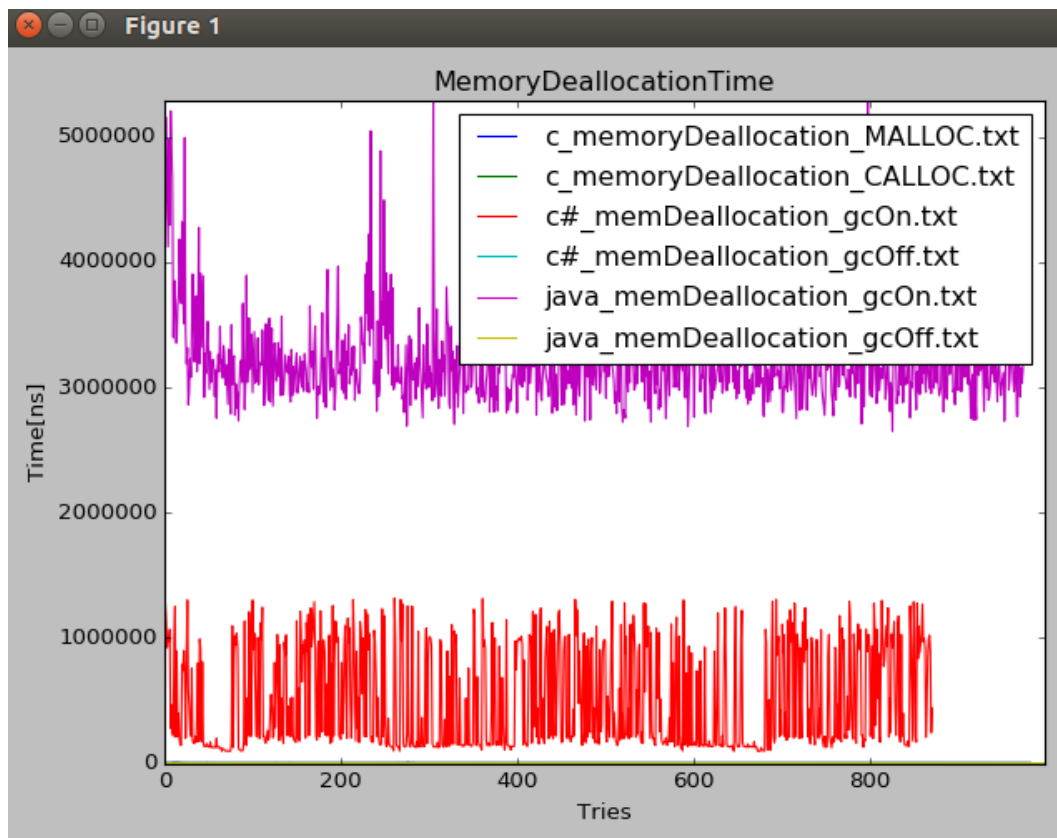


## b. Memory Allocation



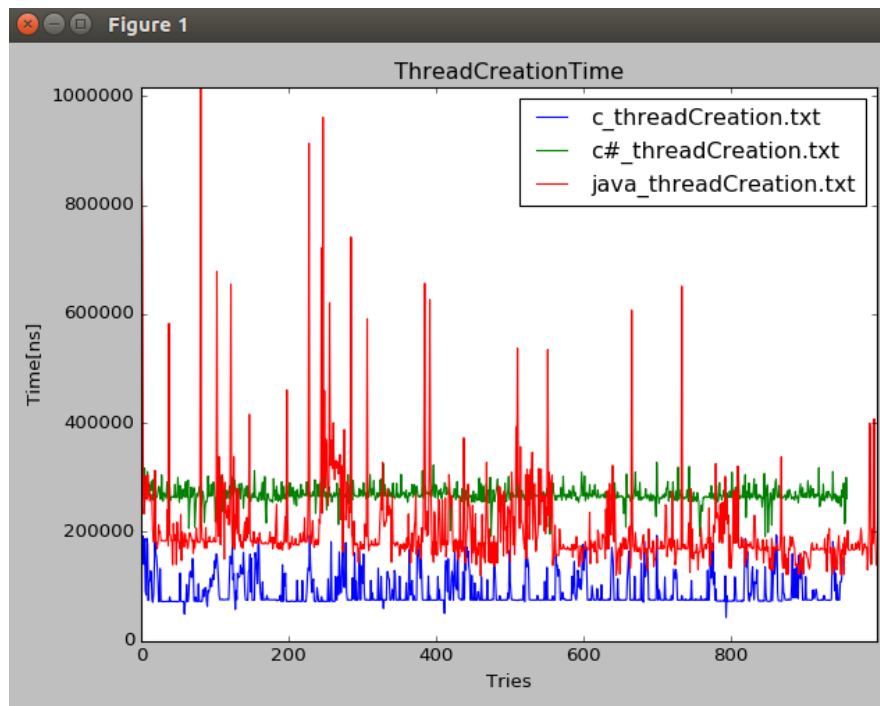
As it can be seen from the plot, the java memory allocation time is significantly greater than the one from C# and C. Moreover, it can be seen that this time is not constant. It varies a lot. The same behavior can be seen in the C# implementation too. However, the C implementation is the most stable one, having quite a constant and really small time for the memory allocation feature with really no significant difference between the use of malloc and calloc.

### c. Memory Deallocation



As it can be seen, again, the Java implementation for the memory deallocation with the garbage collector on is the most time consuming one. We can also see that this implementation is quite expensive and, nevertheless, it is also quite disproportionate. The same behavior can be seen also for the C# implementation with the garbage collector on. However, this one is not quite so expensive. Nevertheless, these two are a lot more time consuming than the java, c# (without garbage collector) and c (with memory allocated with both malloc and calloc) memory deallocation implementation. They are constant and, nevertheless, almost close to 0.

#### d. Thread creation



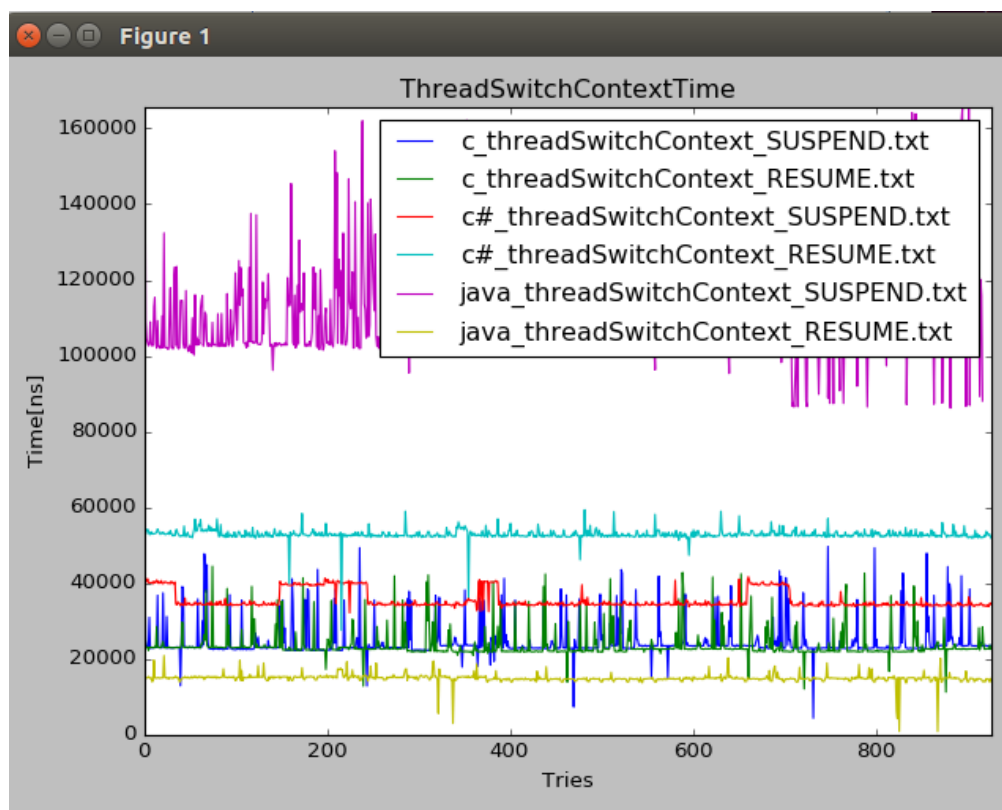
As it can be seen, again, java presents really huge spikes of elapsed time compared to its average time. However, on average, Java performs a lot better regarding thread creation than the C# implementation. Nevertheless, as it can be seen, the one that performs the best is, as before, the C implementation.

#### e. Thread switch context

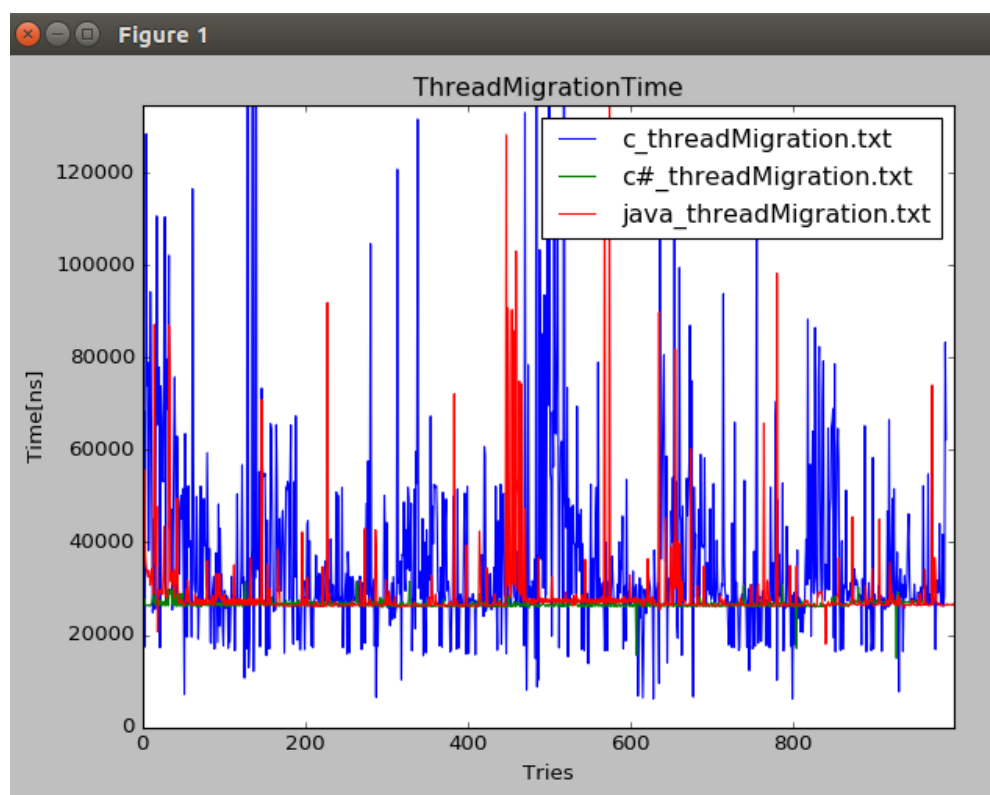
First and foremost, we will talk about the **Suspend** part of the context switch. It can be seen that the Java implementation for this is the most time consuming one. Also, it is not constant, consisting of really multiple and often spikes. However, the C# one is quite constant and significantly lower than the java one. Nevertheless, the C one is the fastest on average (even if it presents spikes).

Secondly, we will talk about the **Resume** part of the context switch. It can be seen that, for this feature, the most time consuming one is the C# one which, nevertheless, is quite constant. However, the C one is faster on average than the C# one, even if it presents spikes. Nevertheless, most surprisingly, the fastest one is the Java implementation, being quite constant on average.





## f. Thread migration



The final feature we have tested is thread migration. It is important to note that the same implementation has been tested from all the three microbenchmarks, either directly or through bridges, but the overhead of the call was not taken into consideration. However, as it can be seen, the results quite vary. On average, all three languages have behaved the same. However, each language introduced spikes, with Java and C being the most significant ones, whilst the C# implementation is quite constant.

## 7. Conclusion

In conclusion, I have developed a modularized application that is capable to test six different features on three different programming languages. It can be accessed through a friendly user interface and all the measurements are displayed as plots. The application has been developed with the purpose of being easily scalable, modifiable and reliable. Thus, an improvement point for this application could be to introduce new features to be tested or even new microbenchmarks implemented in different languages to test different features.

## 8. Bibliography

### a. Java

- [1]. "JNI with C++ : UnsatisfiedLinkError", zobayer1, June 19<sup>th</sup> 2013  
<https://ubuntuforums.org/showthread.php?t=2155683>
- [2]. "Java Magic. Part 4: sun.misc.Unsafe", Mishadoff, February 26<sup>th</sup> 2013  
<http://mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe/>
- [3]. "Unsafe Java Documentation", Oracle  
<http://www.docjar.com/html/api/sun/misc/Unsafe.java.html>
- [4]. "Creating and Starting Java Threads", Jakob Jenkov, October 11<sup>th</sup> 2018  
<http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html>
- [5]. "Java - Thread Control", Tutorials Point  
[https://www.tutorialspoint.com/java/java\\_thread\\_control.htm](https://www.tutorialspoint.com/java/java_thread_control.htm)

### b. C

- [6]. "Linux man pages", Michael Kerrisk  
<https://linux.die.net/man/>

[7]. "POSIX thread (pthread) libraries", Carnegie Mellon University

<https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html>

[8]. "Linux Programmer's Manual", Michael Kerrisk

<http://man7.org/linux/man-pages/man7/pthreads.7.html>

[9]. "How to use sched\_getaffinity and sched\_setaffinity in Linux from C?", Basmah, May 7<sup>th</sup> 2012

<https://stackoverflow.com/questions/10490756/how-to-use-sched-getaffinity-and-sched-setaffinity-in-linux-from-c>

[10]. "Difference between CLOCK\_REALTIME and CLOCK\_MONOTONIC?", NPE, August 19<sup>th</sup> 2010

<https://stackoverflow.com/questions/3523442/difference-between-clock-realtime-and-clock-monotonic>

### c. C#

[11]. "Install Mono", Mono Project

<https://www.mono-project.com/download/stable/#download-lin>

[12]. "C# Tutorial", Tutorials Point

<https://www.tutorialspoint.com/csharp/>

[13]. "How do you convert Stopwatch ticks to nanoseconds, milliseconds and seconds?", Scot Davies, February 24<sup>th</sup> 2010

<https://stackoverflow.com/questions/2329079/how-do-you-convert-stopwatch-ticks-to-nanoseconds-milliseconds-and-seconds>

[14]. "Creating and Using DLL (Class Library) in C#", Anoop Kumar Sharma, October 30<sup>th</sup> 2018

<https://www.c-sharpcorner.com/UploadFile/1e050f/creating-and-using-dll-class-library-in-C-Sharp/>