Logo, company name

Description automatically generated

**Structure of Computer Systems**

*~ Project ~*

*Time execution measurements for processes in different programming languages*

Student: Onaci Andreea-Maria

Group: 30431

Contents

[1. Introduction 3](#_Toc153880007)

[a) Context 3](#_Toc153880008)

[b) Motivation 3](#_Toc153880009)

[c) Objectives 3](#_Toc153880010)

[2. Bibliographic study 3](#_Toc153880011)

[3. Design & Analysis 4](#_Toc153880012)

[Memory Allocation 4](#_Toc153880013)

[Memory Static Access 5](#_Toc153880014)

[Memory Dynamic Access 6](#_Toc153880015)

[Thread Creation and Execution 8](#_Toc153880016)

[Thread Switching Context 9](#_Toc153880017)

[Thread Migration 10](#_Toc153880018)

[4. Implementation 12](#_Toc153880019)

[5. Testing & Validation 13](#_Toc153880020)

[6. Conclusion 13](#_Toc153880021)

# Introduction

## Context

This project explores and measures the execution times of critical computational operations, including memory allocation, memory access, thread creation and execution, thread context switching, and thread migration, across a range of programming languages. By comparing these operations, I aim to provide valuable insights into performance trade-offs and the impact of language selection, runtime environments, and optimization techniques. This project serves as an educational resource for students and developers, aiding in informed language and system process choices for various project requirements.

## Motivation

As a student, my motivation is to gain hands-on experience and a deep understanding of how programming languages, memory management, and multithreading impact software performance. This project provides me an opportunity to measure and compare the execution times of key operations. By doing so, I aim to enhance my knowledge, learn the practical implications of language selection, and prepare myself for informed decision-making in my future projects and coursework. This project empower me to develop a solid foundation in computer science and software development.

## Objectives

This project aims to comprehensively explore the impact of language choice, memory management strategies, and multithreading practices on software performance. My objectives include measuring and comparing the execution times of critical operations, such as memory allocation, memory access, thread creation and execution, thread context switching, and thread migration, across a range of programming languages. I seek to evaluate how popular languages like C, C# and Java perform in these scenarios. Additionally, I'll analyze memory management, the influence of runtimes and optimization techniques, and investigate the trade-offs associated with multithreading. My project's primary goal is to create a resource for developers and practitioners, enabling informed decision-making when selecting languages and system processes for diverse project requirements.

# Bibliographic study

This study synthesizes insights from diverse sources, including GNU's CPU affinity documentation, discussions on mutex and critical sections, pthread library impact on thread context switching, Java concurrency with Countdown Latch, Microsoft's Win32 multithreading, and a Quora discussion on memory allocation speed in Java vs. C/C++. Lecture notes complement these perspectives. The compilation offers a succinct exploration of factors influencing time execution measurements across various programming languages.

<https://www.gnu.org/software/libc/manual/html_node/CPU-Affinity.html>

https://stackoverflow.com/questions/800383/what-is-the-difference-between-mutex-and-critical-section

<https://stackoverflow.com/questions/65272441/thread-context-switching-in-program-output-using-pthread-library>

<https://www.baeldung.com/java-countdown-latch>

<https://learn.microsoft.com/en-us/cpp/parallel/multithreading-with-c-and-win32?view=msvc-170>

<https://www.quora.com/Is-memory-allocation-faster-in-Java-than-in-c-c++-and-why>

Lecture notes

# Design & Analysis

The design of this project revolves around a carefully constructed set of experiments aimed at measuring and analyzing the performance characteristics of C, Java, and C# with a particular focus on time execution, memory access, and multi-threading.

## Memory Allocation

The C code segment dynamically allocates memory for a one-million-element integer array. This process reserves a contiguous block of memory in the computer's heap, sized to accommodate the specified number of integers.

The Java code employs static memory allocation, initiating the allocation process with a precise measurement of time. During this allocation, a dynamic ArrayList is utilized to store integers, although the memory allocation mechanism remains static. This means that the memory size required for the ArrayList is determined at compile-time, and the data structure is fully prepared for use once initialized. The code then reports the time taken for memory allocation, providing valuable insights into the efficiency of this static memory allocation approach.

The C# code snippet demonstrates memory allocation through the creation of an integer array containing one million elements. It employs static memory allocation, where the array size is fixed at compile-time, though the compile time for the C# dynamic memory allocation does not provide a very big difference comparing to the static one. The code showcases the fundamental process of reserving memory space for the array, readying it for data storage, without involving dynamic or runtime allocation procedures.

|  |  |  |  |
| --- | --- | --- | --- |
| Measurement\Language | C | Java | C# |
| Measurement 1 | 0.0208 | 0.0143 | 0.0144 |
| Measurement 2 | 0.0304 | 0.026701 | 0.0091 |
| Measurement 3 | 0.0268 | 0.0139 | 0.014 |
| Measurement 4 | 0.0262 | 0.0126 | 0.0116 |
| Measurement 5 | 0.0259 | 0.013399 | 0.0115 |
| Average: | 0.026 | 0.01618 | 0.0121 |

**C:** In the memory allocation and access tests, we can notice that the C programming language tend to have longer execution times. This suggests that C might not manage memory as efficiently as Java or C#. It resulted in slightly slower performance for the tasks I examined. It's worth noting that the efficiency of C varied depending on the specific operation. Overall, in my tests, C seemed to be the least efficient of the three languages for the operations I tested.

**Java:** Java, on the other hand, consistently demonstrated faster execution times for memory allocation and access operations compared to C. This implies that Java's memory management and execution mechanisms are well-optimized, making it more efficient for these tasks. Notably, Java provided a stable and predictable performance across all measurements. On average, Java outperformed C, making it a solid choice for memory-related tasks.

**C#:** Similarly, C# also displayed swift execution times for memory allocation and access operations, putting it on par with Java. This suggests that C# shares Java's efficiency in memory management and execution. C# exhibited consistent execution times in my measurements, reinforcing its reliability in handling these tasks. On average, C# outperformed C, aligning it with Java as a suitable choice for memory operations.

These findings highlight the efficiency and consistency of both Java and C# in comparison to C for this operation.

## Memory Static Access

In my C implementation, static memory access occurs through a loop that initializes each element of the statically allocated integer array, arrayStaticAccess, with its index value. This operation does not involve dynamic memory allocation and deallocation, making it generally faster than dynamic memory access. The time taken for the static memory access is then printed to the console in milliseconds for analysis, demonstrating the speed at which data can be retrieved from statically allocated memory locations.

In the Java code, memory access is achieved through a statically allocated integer array, arrayStaticAccess, which is pre-allocated with a fixed size of 1,000,000. A loop is used to initialize each element of the array with its respective index value. This process demonstrates static memory access, as the locations in memory are determined at compile-time and do not involve dynamic memory allocation or deallocation. The difference between the recorded timestamps indicates the execution time and reflects the speed at which data can be retrieved from these statically determined memory locations.

In the C# code, memory access is accomplished through a statically allocated integer array, arrayStaticAccess, with a fixed size of 1,000,000. A loop is utilized to initialize each element of the array by assigning it its corresponding index value. This code exemplifies static memory access, as the memory locations are predetermined at compile-time and do not involve dynamic memory allocation or deallocation.

|  |  |  |  |
| --- | --- | --- | --- |
| Measurement\Language | C | Java | C# |
| Measurement 1 | 0.780700 | 1.1694 | 1.2469 |
| Measurement 2 | 0.798700 | 1.3699 | 1.1924 |
| Measurement 3 | 0.884000 | 1.2213 | 1.1474 |
| Measurement 4 | 0.796700 | 1.2435 | 1.0555 |
| Measurement 5 | 0.828600 | 1.1706 | 1.1432 |
| Average: | 0.81774 | 1.2349 | 1.1571 |

**C:** In the realm of memory static access, the C language exhibits the shortest execution times among the three languages, which means that it performs these operations most efficiently. C's consistently lower execution times across all measurements demonstrate its stability in handling memory static access tasks. This indicates that C is a strong choice for operations that require quick memory access and suggests that it excels in providing efficient and reliable static memory access.

**Java:** Java, while showing slightly longer execution times compared to C, still performs memory static access operations efficiently. Its times remain relatively stable across measurements, indicating a consistent performance. On average, Java exhibits slightly longer execution times than C and C# in this specific task. Java remains a reliable and efficient choice for static memory access operations.

**C#:** C# demonstrates execution times similar to Java, falling in the middle ground between C and Java. It exhibits relatively efficient memory static access operations with execution times consistently shorter than Java but longer than C. This suggests that C# provides a stable and competitive performance for static memory access tasks. On average, it falls between C and Java, highlighting its effectiveness and reliability in this context.

## Memory Dynamic Access

The C code initializes a dynamic array structure emulating a list. It starts a timer and iterates through a loop, adding integers from 0 to 999,999 into the dynamically expanding array. The program records the elapsed time using the CPU clock and outputs the time taken to fill the dynamic array in milliseconds. This implementation dynamically reallocates memory when the array reaches its capacity, enabling the addition of elements beyond the initial size. Finally, it frees the allocated memory associated with the dynamic array. This C code demonstrates dynamic memory access at the hardware level by dynamically allocating and accessing memory for an integer array, allowing for runtime memory allocation and efficient data retrieval.

In the Java code, dynamic memory access starts by defining a list, arrayDynamicAccess, and dynamically populating it with integers, increasing its size to 1,000,000 through a loop. This dynamic memory approach differs from static memory allocation since the memory locations are not predetermined at compile-time but rather expanded as needed. This code highlights the efficiency and flexibility of dynamic memory access in Java, where memory is allocated and managed dynamically at runtime. When compiling this specific code, the hardware components, such as the processor, memory (RAM), and storage, play a crucial role in executing the compilation process efficiently and accurately.

In the provided C# code, dynamic memory access is demonstrated by creating a list, arrayDynamicAccess, and populating it with integers in a loop. This approach allows memory allocation and resizing to occur dynamically, as elements are added to the list as needed. However, it's important to note that the focus here is on dynamic memory handling. When compiling this code, the hardware components, including the CPU, RAM, and cache, play a crucial role in managing memory allocation and access, impacting the code's overall performance and efficiency.

|  |  |  |  |
| --- | --- | --- | --- |
| Measurement\Language | C | Java | C# |
| Measurement 1 | 2.452900 | 5.013 | 4.5524 |
| Measurement 2 | 2.419300 | 4.9939 | 4.3828 |
| Measurement 3 | 2.350200 | 4.9853 | 4.5939 |
| Measurement 4 | 2.742400 | 4.7769 | 4.6804 |
| Measurement 5 | 2.456900 | 4.8821 | 4.5204 |
| Average: | 2.48434 | 4.9302 | 4.546 |

**C:** In terms of memory dynamic access, C showcases short execution times, indicating its efficiency for these operations. The consistency of C's execution times across measurements underscores its reliability and predictability in managing dynamic memory access. On average, C outperforms both Java and C# for these operations.

**Java:** Java demonstrates efficient performance for memory dynamic access, employing well-optimized mechanisms for handling dynamic memory access. This results in a solid performance, with relatively consistent execution times across measurements. On average, Java provides reliable and stable performance for dynamic memory access, making it a suitable choice for these operations.

**C#:** C# also performs well in the context of memory dynamic access, offering efficient performance, slightly better than Java’s. This language exhibit well-optimized mechanisms for handling dynamic memory access, resulting in solid and reliable performance. On average, neither Java nor C# significantly outperforms the other for memory dynamic access, positioning them as kinda suitable choices for these operation, because they can not compete with C in this task. The decision between Java and C# may depend on project-specific considerations or personal preferences.

## Thread Creation and Execution

The C code segment focuses on the approach to thread creation and execution. It first initializes a critical section, creates start and done events, and then proceeds to create multiple threads using the \_beginthreadex function. Each thread is assigned the threadFunctions function as its entry point. The code waits for all threads to finish using WaitForMultipleObjects. In this way, it showcases how threads are created and executed concurrently in C. The hardware components relevant to this code execution primarily include the CPU, which manages thread scheduling, and the memory, where thread-related data and function execution contexts are stored.

In this Java code, thread creation and execution are demonstrated. It involves creating a specified number of threads (in this case, one thread) and starting them. Each thread is initialized within a loop, and the start() method is called to initiate their execution. The code then uses the join() method to ensure that each thread completes its execution before proceeding. This approach allows for the creation and orderly execution of threads. Regarding hardware components, when this code is compiled and run, it relies on the underlying hardware to manage and execute the threads efficiently, with the number of threads and their execution timing determined by the hardware's capabilities and scheduling mechanisms.

In this C# code, a specific number of threads (in this case, 1) are created and executed. It initializes an array of threads, starts each thread, and then waits for all threads to complete their execution using the Join method. The code provides a controlled approach to thread creation and execution. When compiling this code, the hardware components, such as the CPU, play a crucial role in managing and scheduling the threads for execution on the available cores. The code's performance can be affected by factors like the number of CPU cores, thread scheduling policies, and available system resources.

|  |  |  |  |
| --- | --- | --- | --- |
| Measurement\Language | C | Java | C# |
| Measurement 1 | 0.444400 | 0.5128 | 0.3536 |
| Measurement 2 | 0.522300 | 0.5191 | 0.3451 |
| Measurement 3 | 0.517300 | 0.4316 | 0.3108 |
| Measurement 4 | 0.596800 | 0.427 | 0.3156 |
| Measurement 5 | 0.439100 | 0.4438 | 0.3218 |
| Average: | 0.50398 | 0.4669 | 0.3294 |

**C:** When it comes to thread creation and execution, C exhibits longer execution times compared to Java and C#. This suggests that C may have less efficient mechanisms for handling threads, leading to slightly slower performance for these specific task. The variations in execution times across measurements indicate that C's efficiency fluctuates depending on the specific operation. On average, C lags behind Java and C#, making it the least performant among the three languages for the operations considered.

**Java:** Java demonstrates efficient thread creation and execution, with execution times falling between C and C#. This implies that Java's mechanisms for managing threads are reasonably optimized, resulting in a solid performance for these tasks. Furthermore, the relatively consistent execution times across measurements suggest that Java provides stable and predictable performance for thread-related operations. On average, while Java doesn't outperform C# and is slightly slower than C, it remains a reliable choice for tasks involving thread creation and execution.

**C#:** C# outperforms both C and Java in terms of thread creation and execution, offering significantly shorter execution times. This indicates that C# shares Java's efficiency in managing threads and execution, and in fact, exceeds it. The stable execution times across measurements reinforce the reliability of C# in handling these tasks. On average, C# proves to be the top choice for tasks involving thread creation and execution operations, standing out as the most efficient language among the three.

## Thread Switching Context

The provided C code illustrates a thread context switching approach. Each thread, represented by the ThreadFunctionSwitchingContext, enters a critical section after a flag is set to 1, allowing multiple threads to run concurrently. The code then creates multiple threads, each executing the ThreadFunctionSwitchingContext, and the flag is set to initiate the context switching. The code waits for all threads to complete using WaitForMultipleObjects. Regarding the hardware components, when this code is compiled and executed, it relies on the hardware components of the system, including the CPU, memory, and other resources, to facilitate thread execution and context switching efficiently.

The Java code snippet initiates a thread context switch operation to simulate concurrent task management. It creates two threads using the Thread class and uses CountDownLatch to synchronize their execution. The threads wait for a start signal, which is then released to begin their execution, and they signal completion once they're done. Regarding the hardware components, the efficiency of this code may depend on the number of processor cores and the operating system's ability to manage thread context switching efficiently, as these factors play a significant role in determining the effectiveness of concurrent execution and context switching performance.

The provided C# code demonstrates thread context switching. Each thread, defined by the ThreadFunctionSwitchingContext function, waits for a flag to become true while utilizing synchronization constructs. When the flag is set, it indicates the beginning of context switching. Within a multi-threaded environment, this process allows different threads to take turns executing and sharing the CPU's processing time. The code creates multiple threads, each executing the context switching function, and coordinates their execution through events and synchronization using the EnterCriticalSection and LeaveCriticalSection functions. Regarding hardware components, this code operates within a multi-core or multi-processor system, as it utilizes multiple threads to simulate context switching. Modern processors with multiple cores can efficiently handle multiple threads, enabling parallel execution and context switching among them. This code leverages these hardware capabilities for simulating context switching behavior.

|  |  |  |  |
| --- | --- | --- | --- |
| Measurement\Language | C | Java | C# |
| Measurement 1 | 0.001000 | 0.3292 | 0.2367 |
| Measurement 2 | 0.000700 | 0.2007 | 0.1868 |
| Measurement 3 | 0.000800 | 0.2125 | 0.1415 |
| Measurement 4 | 0.000700 | 0.2633 | 0.1559 |
| Measurement 5 | 0.001000 | 0.2034 | 0.1709 |
| Average: | 0.00084 | 0.2418 | 0.1784 |

**C:** In the case of thread context switching, C exhibits impressively short execution times compared to Java and C#. These results suggest that C may employ highly efficient mechanisms for managing thread context switches, resulting in exceptionally fast performance for these specific tasks. The consistency of C's execution times across measurements demonstrates its reliability and predictability in handling thread context switching. On average, C significantly outperforms both Java and C, positioning it as the most efficient choice for these operations among the three languages.

**Java:** Java shows longer execution times for thread context switching in comparison to C and C#. This implies that Java's mechanisms for managing thread context switches are reasonably optimized, but not as good as C’s and C#’s. The relatively consistent execution times across measurements indicate that Java delivers reliable and stable performance for thread context switching. On average, Java maintains a performance edge that falls behind C and C#, making it the less good choice for this operation.

**C#:** C# also performs well in the context of thread context switching, with execution times between Java’s and C, but much closer to Java’s. This suggests that C# shares efficiency in managing thread context switches. The stable execution times across measurements reinforce the reliability of C# in handling these tasks. On average, C# exhibits longer execution times than C, but better than Java for thread context switching.

These findings underscore the efficiency and consistency of C compared to both C# and Java for the operation examined.

## Thread Migration

The C code demonstrates a thread migration approach. In this code, a new thread, ThreadFunctionMigration, is created and assigned an affinity mask to run on a specific CPU core. Once the migration is complete, the code prints the migration time in milliseconds. Regarding hardware components, this code's behavior and results depend on the specific hardware configuration, including the number of CPU cores, their capabilities, and the operating system's thread management. The code's execution may yield different results on systems with varying hardware configurations.

In my Java code, a specific thread migration approach is employed. The code initializes a single thread and uses a CountDownLatch to coordinate its execution. The thread enters a loop that repeatedly checks a shared flag variable, and upon a change, it records the migration time. The flag variable is linked to the thread's index, and each thread has a unique index. Once the migration time is calculated and displayed, the code awaits the completion of all threads. This approach showcases how threads can be coordinated and monitored in a multi-threaded environment. Regarding hardware components, when compiling this specific code, the behavior and performance can be influenced by factors such as the number of CPU cores, memory, and scheduling mechanisms on the host machine, as they impact the thread migration process and execution speed.

In the C# code, a thread migration approach is demonstrated. It employs the Task.Run method to create multiple threads, each simulating a thread migration scenario. These threads continuously check a shared integer, threadNumber, and if it's not equal to their thread identifier, they exit, simulating a context migration. It showcases how threads can be used to simulate context switching and highlights the potential for thread migration in concurrent programming. When compiling this code, hardware components like the CPU and memory architecture play a crucial role in determining how efficiently context switches and thread migrations are handled, as these operations are influenced by the hardware's thread management capabilities and memory access speeds.

|  |  |  |  |
| --- | --- | --- | --- |
| Measurement\Language | C | Java | C# |
| Measurement 1 | 0.018300 | 0.2065 | 0.0437 |
| Measurement 2 | 0.015300 | 0.2063 | 0.0427 |
| Measurement 3 | 0.027400 | 0.2215 | 0.0425 |
| Measurement 4 | 0.015000 | 0.1774 | 0.0426 |
| Measurement 5 | 0.030600 | 0.1906 | 0.0496 |
| Average: | 0.02132 | 0.2005 | 0.0442 |

**C:** In the context of thread migration, C exhibits the shortest execution times among the three languages, making it the most efficient for these operations. These results suggest that C's mechanisms for handling thread migration are highly optimized, resulting in very fast performance for these specific tasks. The consistency of C's execution times across measurements underscores its reliability and predictability in managing thread migration. On average, C significantly outperforms both Java and C#, establishing itself as the most efficient choice for these operations.

**Java:** Java, while showing longer execution times compared to both C and C#, still maintains its efficiency in managing thread migration. This implies that Java's mechanisms for handling thread migration are reasonably optimized, resulting in a solid performance for these tasks. The relatively consistent execution times across measurements highlight Java's reliability and stability in managing thread migration. On average, Java remains a suitable choice for thread migration, with a performance behind C and C#.

**C#:** C# performs well in the context of thread migration, with execution times slower than C’s ones. This indicates that C# shares similar efficiency in managing thread migration. The stable execution times across measurements reinforce the reliability of C# in handling these tasks. On average, C# lags behind C but remains a suitable choice for thread migration, being better than Java.

These findings underscore the efficiency and consistency of both C# and C compared to Java for the operation examined.

# Implementation

In my project, I employed Windows-specific time execution measurement techniques to analyze various aspects of software performance. For memory allocation and static/dynamic memory access, I utilized C, Java, and C# to measure execution times. In C, I used the QueryPerformanceCounter function, while Java featured thread creation and synchronization using CountDownLatch, and C# employed Stopwatch. Additionally, I evaluated thread context switching and migration. These techniques ensured comprehensive performance assessment across different programming languages and operations.

My choice of C, C#, and Java for performance comparison was well-suited to my project's goal of measuring execution time in various scenarios. C provided low-level memory control for precise memory allocation and access measurements. Java, with its platform-independent nature, allowed for thread creation and synchronization analysis. Meanwhile, C# offered a balance between high-level programming and efficient memory access measurement capabilities. This selection facilitated a holistic evaluation of performance aspects crucial for various software development scenarios, ensuring a comprehensive and informative comparison across the chosen programming languages.

To successfully run the project, the user requires a computer system with suitable hardware specifications. The specific hardware requirements may vary depending on the complexity of the processes and languages being tested, but a general guideline includes having a modern multi-core processor, sufficient RAM (8GB or more), and ample storage for data and development tools. Additionally, a solid-state drive (SSD) is recommended for faster data access, which is crucial for precise time measurements. A reliable internet connection may be necessary for downloading and updating software tools. Finally, to ensure consistency and accuracy in your measurements, having a stable and well-maintained hardware environment is essential.

Along the implementation of this project, several challenges and limitations were encountered. Challenges included addressing language-specific intricacies and optimizing code for accurate time measurements across C, C#, and Java. Additionally, variations in hardware configurations among the test systems sometimes introduced discrepancies in the results. Limitations encompassed the scope of the project, which focused on specific use cases, and may not fully represent all possible scenarios in software development. Moreover, while the findings offer valuable insights into these selected areas, their generalizability to broader contexts and diverse programming tasks should be approached with caution, as the intricacies of software development are vast and multifaceted.

# Testing & Validation

In this project, the experiments were conducted on an AMD Ryzen 9 processor featuring 8 cores and a base clock speed of 3.30 GHz, complemented by 16 GB of RAM and a 512 KB cache. The choice of this hardware configuration was pivotal in ensuring reliable and consistent testing across the various programming languages and processes under investigation. The substantial 16 GB of RAM capacity offered ample memory resources, while the 512 KB cache contributed to efficient data access. Furthermore, the project was executed within a Windows 10 operating system environment, further influencing the testing environment. These hardware and software specifications collectively played a central role in facilitating parallel execution, aligning with the project's focus on thread management and execution time measurements.

# Conclusion

In conclusion, this project has shed light on the nuanced aspects of time execution in C, C#, and Java across a range of crucial software development processes. My findings have demonstrated that language choice can significantly impact the efficiency of memory allocation, static and dynamic memory access, thread creation, and thread context switching and migration. The observed variations provide valuable insights for developers aiming to optimize code and make informed decisions when selecting the appropriate language for specific tasks. However, my project encountered challenges related to language intricacies and hardware variations, and the project's scope is limited to specific use cases. As I look to the future, further research could explore additional programming languages and a broader spectrum of software development scenarios. Nevertheless, my project contributes to the ongoing discourse in software development and underscores the importance of considering time execution measurements in language selection and code optimization, offering practical guidance for developers and organizations seeking enhanced software performance and efficiency.