Bachelor Thesis

Field of study

Technische Informatik

Florian Pîrvu

Thesis Title: C++ Crossplatform Bibliothek zur Performanceanalyse
von Anwedungen unter dynamisch generierten Lasten

First examiner: Prof. Dr. Thomas Kirchmeier

Second examiner: Prof. Dr. Hubert Högl

Submission date: 20.01.2022

Bachelor Thesis author
Florian Pîrvu
Asternweg 2
86399 Bobingen
Tel +49 176 3693 7974
pflorian306@gmail.com

Contents

1. Introduction

1.1 Motivation

1.1.1 Choosing the right tools

When the computer was first invented, the world was much simpler and at the same time much rougher. The computers back then understood only a limited number of instructions and the tools that one needed to learn in order to conquer these new machines were unsophisticated (regarding the today's standard). At the same time people were restrained by the small size of memory and the lack of freedom when it came to working with computers. Back in the days, a typical computer was for example the *Compaq Presario 425*, which had a 25MHz Intel 486 CPU (so it could execute 25 000 000 instructions per second) and 200MiB hard disk[1][1]. As time passed engineers around the world worked together and upgraded the gadget and each year they increased the number of commands and the memory of computers and so created the machine to what it is today known as a *personal computer*(short PC). The new sets of instructions were also made differently in order to satisfy distinctive needs and in time they developed themselves into programming languages known today.

Despite the numerous amount of languages, not all of them are used. A study made in 2020 researched 50 projects of an open source project management platform called *Gitee*, showing which programming language is likely to be more popular based on multiple factors such as the popularity and attention the projects were getting on the platform. The results were split in five categorizes as showing in fig. 1.1. As observed the languages that maintain their popularity over the time are C#, Java, C++, Python and JavaScript[2].

Showing this fact one could wonder, why even bother using the complicated C++, when there are other more easier options like Java or Python. The answer to this question is simple, it's all about performance.

A paper published in 2006 stated that when it came to performance such as memory for example, C++ was far superior in comparison to Java and mentioned that modern languages like Java or C# hide their complexity under the hood. To prove this statement the author coded himself "a program that manipulated half a billion objects. Its C++ implementation required 3 Gbytes of real memory to run. A Java implementation would easily need that amount of memory just to store the objects'housekeeping data". He also added that the language, although complicated with its huge size and complexity,

---

[1]For users not familiar with memory, as of April 2021, the size of the game *Call of Duty: Modern Warfare* is around 230GiB. That is circa 1000 times bigger than the hard drive of *Compaq Presario 425*

category	Name	Programming language
1	Stable popular languages	C#, Java, C++, Python, JavaScript
2	Unstable popular languages	C, PHP, Lua, Objective-C, Shell
3	Common languages in emerging fields	ActionScript, Dart, CSS/HTML, TypeScript
4	Common languages in traditional fields	Arduino, Kotlin, VHDL, Swift, Rust, Scala
5	Potentially super popular language	Go

Figure 1.1: Classification Results of Programming Languages

comes as a combination of "extreme efficiency and expressiveness". Nevertheless Spinellis commented that for smaller devices such as an MCU[2] or systems code, using the C language is the way to go[3].

This lead me to the conclusion, that in order to still have a newer interface using OOP[3] and at the same time compatibility with the efficient C language, in which many operating systems are coded, C++ is the best candidate of them all. However performance is a vague term and can describe a lot of things, that's why, to better understand the purpose of this project, one should get a glimpse at *Performance* when it comes to computer science.

1.1.2 Performance & Testing

Performance of software is the process of analyzing a given program and reacting to problems that may or may not occur during runtime. The less unexpected problems emerge, the better the performance of that software[4]. To make sure fewer problems occur, one could adopt methods like the *Software Performance Engineering*(short SPE), which uses quantitative techniques to identify designs with lesser flaws and so spare the developers significant time in their implementation[5].

An idealistic scenario would be, if the product would have no flaws at all. But as many know, perfectness is hard to achieve, even harder if people keep developing the product. Even if the outcome may be perfect at a certain point in time, the continuous development will certainly show incompatibilities with the functional state of the product.

<div align="center">

"If anything can go wrong, it will"

</div>

<div align="right">

Murphy's law

</div>

To avoid the release of a defect product and support the *SPE* approach, one needs to implement additional tests, because bugs can appear even after deployment as the development process advances.

Testing in the IT-Branch is a process with the main goal of finding bugs in a software program or application. These bugs refer to errors or faults and can occur because of a bad command sequence specified in the program's source code or incompatibility of other member of the program. This can often occur especially when the "amount of libraries are high and the mutual dependencies complex"[6]. A successful test can be achieved when its main requirements are met. Some requirement examples would be the execution on different environments, time constraints or delivering expected outputs for randomly chosen

---

[2]Microcontroller Unit
[3]Object Oriented Programming

inputs. Conditions are set by the tester and may vary. These are frequently chosen based on user cases and logical expectations.

1.1.3 General

Most companies put a lot of effort in delivering high performance products to their customers. In order to do so, each of them test their gadgets, machines or software for possible failure scenarios. The test phases usually take place before the release of a new product or the deployment of an update that brings new features to an existing product.
Now-a-days tests are being fully automated, which decreases the failure possibility that can happen because of human errors[7]. Unfortunately the creation of fully automated tests is not as easy as it may sound. Many testing developers know the struggle of finding the right tools for the job without filling their projects with unnecessary dependencies that will overload the program and occupy valuable memory.

"As ironic as it seems, the challenge of a tester is to test as little as possible. Test less, but test smarter"

Federico Teldo, Co-Founder Abstracta US

Additionally the problem enhances when a company reaches a certain size with a significant number of customers, which tend to run the product on different machines and architectures. In order to keep their customers satisfied, producers need to adapt their products to support newer or older machines. This makes software analysis even more difficult because of the increasing complexity, which comes with different systems. For this reason many companies dedicated themselves to creating programs that focus only on software examination and fixing bugs that may occur during an analysis. In time a new trend has been created with demands so high that rapidly developed itself into a new market section.

1.2 Growing markets

For the last decade the software testing market has been developing and now its popularity increases by the day. According to "Global Market Insights" the testing software market has grown up to 40 billion dollars by the end of 2020 and is predicted to grow up to 60 billion dollars in the next 6 years as shown in fig. 1.2.[8]
Multiple scientific papers and studies enforce this statement with different statistics of industries, which started adapting and reacting to this trend using the model "EaaS" which stands for *Everything as a service* and created *Testing as a Service*(TaaS). With this model customers not only pay for the current state of a product, but also for a service subscription, where they get updates and new features for the specific product and additional support from the company. The advantages that benefit the customer are set by the company for each of their subscription. The basic rule is that you get more accurate results if you pay more. This service usually targets three groups: Developers, End Users and Certification Services.[9]. With the growing popularity of these services, companies quickly got blind sided by the convenience of paying for experts with system knowledge and having them create software specific tests instead of developing them on their own. In reality they would get disappointed by the advantages and cost comparison of putting in the effort to create testing methods specific for their own products.
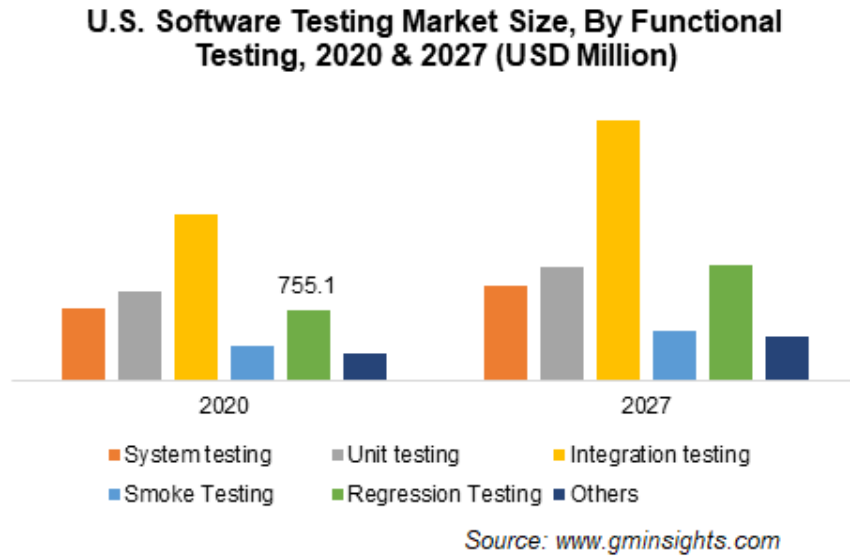
Figure 1.2: US Software Market Size[8]

1.2.1 The costs of perfection

Firms often tend to overlook costs of testing tasks that are insignificant in comparison to the total price of the project. However if these costs are reoccuring, their total price can go up to 40% of the total project's costs[10].

It was proven that the cost of finding an error is about $50 on average [11], and is said that fixing an error after the software was released is four times more expensive than compared to if it was found during the testing phase[12]. That amount of money covers not only for direct testing, which includes the staff, system and program testing, resources, computer time, etc. , but also for indirect testing, which refers to actions that take place because of poor direct testing, like rewriting code, additional analysis meetings or debugging.

There are two common ways to cope with this problem. First, one could use a third party software that specializes in testing other companies's software. This has its advantages, because the tests are already written to work on most compilers for a majority of programming languages and the continuous support and development through customer feedback. The bad part though, is the pricing (which may or may not be a problem for a company depending on its size), the code's ownership, additional privacy issues (because we can't look behind the curtains of a compiled software, therefore we cannot tell what kind of data the program stores about the user or whom may it send it to) and product based implementation.

The second way it would be to let the company's developers test themselves. This method proves often quite efficient, because they are the ones who programmed the software, therefore the best candidates to repair possible bugs and the company is not exposed to any additional privacy risks. However the whole process needs to be led by someone experienced enough to correctly organize current testing tasks (but also future ones), the staff management and avoid expensive commitments such as unnecessary working time and budget. At the begging, in house development may be even more expensive than buying a testing software, but in the long run it will be cheaper as the testing process gets more and more refined.

To avoid these financial expenses, companies can train their developers to consider failure scenarios of the product early in the development stage.

1.2.2 Common approaches

In order to deliver a defect-free product, managers create testing strategies. To develop the most suitable strategy, they must identify the key components for it. These can be identified mostly by answering the following questions adapted from a review of software testing approaches in object-oriented and aspect-oriented systems published in Singapore [12]:

1. Is the objective clear specified?

2. What tools will be used?

3. Is the system fully/partially automated?

4. How will the test benefit the project?

Additionally, the results and procedures need to be recorded in order to make testing phases smoother, because the similar or even the same errors may occur in the future.

1.3 Benchmarking

In addition to testing, companies also have the desire to reach as many customers as possible. In order to do that, many tend to create benchmarks that show the product's superiority in comparison to other products that fulfill the same task and reduce production costs.

According to the *Standard Performance Evaluation Corporation* (short *SPEC*), a benchmark is a "test, or set of tests, designed to compare the performance of one computer system against the performance of others"[13].

A method developed by *Infineon Technologies AG* ,written by Lukas Klaus, explains benchmarking as "The intention [...] [to] keep cost of test constant relative to overall cost of goods sold". Klaus mentioned that once the yearly costs of production were established, one can come up with a financial plan that compares the manufacturing expenses with the "hypothetical best case if all products in question were at benchmark performance". The method can additionally create an indicator, which can be calculated by dividing test costs by the revenue of sold goods[14]. This can be used to help decisions on which part is worth improving in order to create the best product while maximizing customer satisfaction.

The way a benchmark is forged says a lot about its credibility. There are good and bad benchmarks out there, which raises the question "What makes a good benchmark?".

A paper written by Maya Daneva of Institute of Business Information Systems at the University Of Saarland answered this explaining the designs and usage of software benchmarking. According to her the targets of this approach can be classified in different classes of organizations with different set of goals. The most relevant for this thesis are software developers, distributors, testing laboratories and researchers, as well as certifications offices, academic intuitions and independent benchmarking companies. For a well made benchmark, Daneva introduced six crucial criterias[15]:

1. Pertinence

2. Functional coverage

3. Understandability

4. Ease of use

5. Interpretability

6. Reproducibility

When creating the benchmark one would try to achieve perfectness in every aspect of each criteria, but the reality often tends to disappoint our expectations. To explain this in detail you could think about these criteria as a hexagon with six angles. When the developer tries to implement one side of the hexagon to perfection he often would have to give up on other sides as pictured in Figure 1.3. Most of the times one needs to identify the most important aspect that matters the most to him and choose it over less relevant ones (though this doesn't have to be always the case)
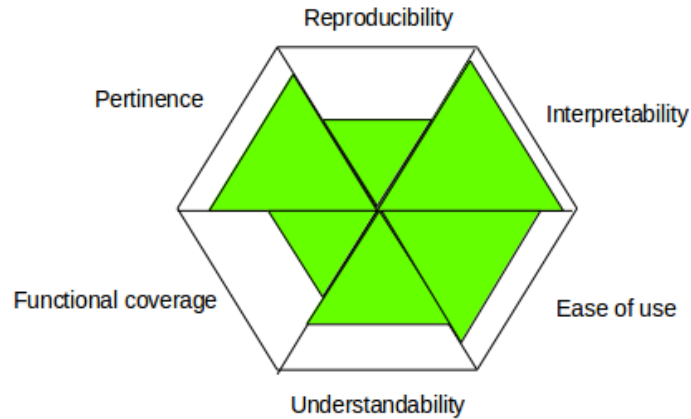

Figure 1.3: Hexagon of criteria

In order to generate reliable statistics, the creator also needs to considerate the additional workload the benchmark is creating and to differentiate between the actual strains put on the system by the application at hand and those created by the benchmark.

1.3.1   Workload

The definition for workload varies from field to field. In the IT-branch it is defined as a unit of measure for your CPU (mostly in %). This tells the user how well his system can handle the number of current running processes.
Workloads are made out of two parts: the executable part and the non-executable part. The executable part refers to code that is being executed when an application software is being executed. The non-executable part refers to the workload produced in order for the executable part to have "a well-defined and reproducible manner". One can also differentiate between natural and artificial workload. Natural workloads are produced by software executing useful tasks and benchmarks that deliver statistics based on such workloads are also called *natural benchmarks*. Artificial workloads however are generated by code that tries to mimic real workload (most time irrelevant code like the incrementation of a variable in a loop). Most benchmarks often implement artificial workloads ,because they "allow[ing] one to measure the limits of a system, or a selected part of it, under different configurations and workloads", while natural ones often tend to need user input, cannot be entirely reproduced and are usually unfair to other processes, based on the application's high optimization for the software to run as smooth as possible, which can put pressure on the system's hardware[16]. When it comes to performance both artificial and real workloads are similar. Additionally good benchmarks that implement artificial workload need to prevent critical situations when the user might overload the system and bring it to a

halt, which for most cases is undesirable.

Many systems calculate their workload over a defined period of time. The following example explains how one system can estimate the workload of a simple application.

Let's say a user starts a calculator program at time $t_0 = 0$. The application will finish initializing a GUI at time $t_{wait} = 2s$ and then an interactive visual window will appear, which will wait for the user to enter some equation. After the equation was typed at $t_{input} = 5s$ and the "Enter"-key was pressed, the application proceeds calculating and delivers the answer at $t_{done} = 6s$. So the CPU-time of the application is 3s(GUI initialization and calculation time). In order to get its workload, the system has to divide this time by the time the system needed for the whole process and multiply it by one hundred to have the result as a percentage:

$$workload = \frac{3s(t_{wait} + t_{calc})}{6s} * 100 \qquad (1.1)$$

A detailed explanation on how to get the values of process and system specific times will follow in section 3.2.2.

1.4 Vision and Objective

1.4.1 Vision

The purpose of this library is to help C++ developers implement their own stress tests and run simulations for their own applications/software with different workloads, process priorities and schedulings, which can be used in different manners to achieve real-time applications and create benchmarks for new products, while reducing the risks of in-house development mentioned in section 1.2.1.

1.4.2 Objective

This thesis defines and explains the library's pillars. Its goal is to create a reliable artificial workload, deliver statistics comparable with already existent tools for Linux and Windows computers on which the software is running with additional easy-to-understand and intuitive methods for handling priorities and schedulers, so that even users with very little knowledge about their operating system can use them. At last it needs to show evidence of reaching the previously mentioned goals and the behavior of the calling machine in a normal and critical state.

1.4.3 Obstacles

Because of the lack of feedback, access to minimal needed hardware and experience in the software development industry, the following points may need further developing:

1. Error handling

2. MacOS and ARM support

3. Better naming conventions

2. State of the art

This chapter summarizes already existing technologies and implementations that are relevant to the library. Despite the fact that most of them were written in in the past using the C language, they are still relevant as they describe how many operating systems work and where many C++ standards come from.

2.1 Current Situation

At the time of writing this (December 2021), there are three mostly used architectures in the IT-Branch. These are Windows, Linux and MacOS. Linux was developed with the UNIX system as its core, while MacOS is only based on UNIX, which means that these are similar but not entirely compatible with each other. The big difference comes with Windows.

Each operating system can deliver informations about its own computer. For that, each of them has its own unique program. Windows uses the "Task-Manger" which comes with a GUI and shows real-time information about the CPU and memory of the computer, it comes with a list of processes and makes it possible for the user to manage them with only a couple of clicks.

Linux on the other hand is more text oriented. This operating system comes with a built in command called "top". This also delivers informations about your system, but it doesn't allow you to manipulate processes like "Task-manger" does. Although not very practicable for developers, these tools allow the user to measure performance in a normal state where your computer is not put under pressure, whereas the most interesting state is when the CPU needs to do a lot of operations and it needs to share its valuable time with other processes. To force such situations, the user can use online stress tests, which use the browser as an workload source, which for many is not very practicable, because it uses an additional program, that runs in the background. Online stress tests rely on internet to work and a stable ethernet or wifi connection to work. This way the results can be very inaccurate if the the network is under pressure.

There are already some alternative libraries on the internet to recreate this method, but most of them are written in other programming languages. Adding them to a C++ Project would rise the complexity of the source code.

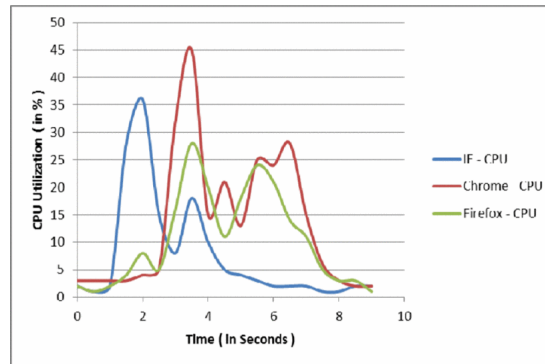Some people would think "Well C++ is a new and modern language. There must be

---

Figure 2.1: Browser CPU utilization comparison[17]

something similar out there". This statement is true and because of its young age there are a limited number of officially tested methods. You could also take a look at the headers implemented in the C++ standard library [18] to familiarize yourself with common algorithms and data structures. Not all operating systems will be covered in this thesis. The following statements are true for the testing machines used in chapter 4.

2.2   Processes

To many operating systems, a process is like a wrapper defined by the kernel in order to allocate resources to an executing program.
When a process is created the system assigns him a *process unique identifier* also called PID(a positive integer). Each process has its own PID, so two different executing programs cannot have the same PID [1]. The methods used to create a new process differ on each operating systems. On Windows you can create processes by calling the `CreateProcessA()` function [19], which returns a HANDLE (the equivalent PID for windows systems) and on linux you can use `fork()`[20]. Each operating system calls one these methods internally every time the computer or the user starts a routine of execution, like starting a service, or a program (browser, spotify, etc). You can think of this system like a binary tree with one or more children with the kernel on top. The children have also an additional attribute called PPID, that contains the PID of its creator. If this is specified to zero then the kernel is the parent.[21][2].

2.2.1   Threads

Each process has at least one thread of execution called the main thread, which (as the name says) contains the `main()` function, that defines the behavior of that process. The main thread can also create other threads that work independently from one another. On UNIX systems one would use the `pthread_create()` and on windows `Create_Thread()`. Once a thread has been created, the main thread can wait for it to finish by calling a method called `join()` and then resume execution or if the created thread called `detach()`, so it will separate itself from the main thread. This way the main thread doesn't have to wait for it to terminate and when a detached thread exists, all of his allocated resources will automatically be freed.
Like processes, threads also have unique identifiers called *Thread identifiers* or TID and OS-specific methods to create them. But the role of this library is to make this whole process as transparent as possible, so we will use the C++ Standard Library's threads (std::thread) and "detach" ourselves from the other ones.

---

[1] On UNIX like machines the first process to be called is the init process with PID 1
[2] There a very few processes that have the PPID=0 (ex. init)

Processes have their own stack, so they can't communicate with each other. This is a huge problem when it comes concurrency (explained in section 2.4), but threads share the stack of their creator-process.

2.2.2 Differences between threads and processes

Many people tend to think of threads and processes as being the same, but they are quite different. First, as mentioned above, threads share the same stack of the creator-process, while processes need intercommunication tools like pipes to talk to each other. Another difference is that a process can have multiple threads attached to it, but a thread cannot belong to more than one process (but this doesn't stop a thread to call `fork()` and so create another process).[3]. Their identifiers are also independent from their creator's. This way a TID can be equal to its creator's (or another process's) PID. For example if a process has its PID equal to 2033, there is no rule that will stop the thread having the same number as its TID.

To summarize and simplify the whole concept, one can imagine a process like an octopus and the threads being it's arms. One octopus has many arms that can execute multiple tasks at the same time, but an arm cannot belong to more than one octopus.

In this library we use multiple threads to simulate a user-specific workload[4], because this way we can time their execution start point with only one shared variable and we don't have to worry about interprocess communications.

2.2.3 Attributes

Normally when we would create a threads using the unix methods, we would pass a pointer to a structure that describes the attributes for that specific thread(that structure can be created with `pthread_attr_init()` and be destroyed with `pthread_attr_destroy()`). Some of these attributes include the scheduling priority, scheduling policy and stack size, which are important for out tests. Unfortunately the standard library doesn't have this option. In order to set and get this attributes we need to use the architecture's dependent functions. In the following sections the attributes of a thread will be discussed.

2.3 Stack Size

The stack is a piece of memory where meta-data and local variables are saved when the `main()` function calls a routine/method. This memory segment is limited and doesn't allow an infinite number of data segments (also called stack frames) being stored in it. The stack uses assembly instruction like `pop` to delete a frame and `push` to add a frame. For consistency the stack will always pop the last element pushed. This is also known as "Last in First Out"[22].

On UNIX systems we would create an attribute structure and pass the desired op-

Figure 2.2: Stack segment
[22]

tions there. However, because we don't use the unix's system function `pthread_create()` we also cannot use the attribute structure. Furthermore the standard library doesn't
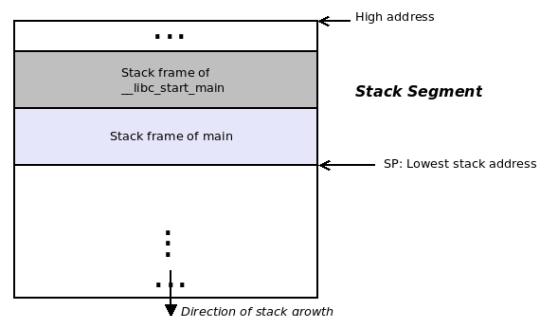
---

[3]If the execution method of a thread calls `fork()` then that process's PPID will be the PID of thread's creator

[4]The user can set the wished workload with a simple function

support such tweaks. This is very important if someone wants to use this for an ARM architecture, because he won't be able to define a meaningful stack size. This doesn't pose any threats for many operating systems out there, but for ARM, which has a limited stack size, can be problematic. For windows this attribute can be set using the `Create_Thread()` function just like in unix using `pthread_create()`.

2.4   Concurrency

Concurrency means that two or more things are happening simultaneously. This pehnomenon is happening everyday almost everywhere we look. Even we as humans are capable of such thing, for example walking and talking at the same time. In computer science concurrency means that more than one process can be executed at the same time. Many systems have this ability, because most of them are multiprocessor computers. Nowadays many systems measure the concurrency of a system by its number of hardware threads. This unit of measure tells us how many independent tasks can the processor handle. Even some single core computers can handle concurrency to some extent. One calculation unit can handle one task at a time, but it can quickly switch to another task if necessary. This is why sometime even single core units give the impression of resolving jobs simultaneously[23, Chapter 1]. Although performant, this method does not come without its flaws. When a cpu gets a new task, the resources of the old task (eg. local variable, meta data of the current task, etc.)will be replaced by those of the new job. This is also known as "Context Switching".

2.4.0.1   Context switching

Context switching can be triggered with an interrupt or a syscall. One example would be when another thread/process with a higher priority starts. This can happen, because the system would have to execute `fork()` for processes or `clone()` for threads, which are also system calls. This would trigger the cpu to reschedule his task based on the schedule and priority of the tasks in the tasks list[24]. The reordering of tasks after switching CPUs will be explained in detail in 2.6.2.3.

2.5   CPU Affinity

The affinity of a CPU determines the number of processors that one process can use for its threads. This library offers methods to restrict the number of CPUs off which a process can run on. This is sometimes desirable because of the following reasons:

1. Data invalidation: When a process starts, the user cannot tell from outside on which CPU that thread started. When a process finishes its time-slice, he has to give up its CPU for others to use it and it come back later, but it won't necessarily start on the same CPU as the last time [5]. When this happens, entries in that CPU's cached must be replace or removed, which is known as "Cache invalidation". This is not a flawless method and cache inconsistencies can appear.

2. Emergency CPU: On real-time systems, where human lives are at risks, many developers will deliberately block some CPUs (on a multicore machine) to use them when the system returns erros and needs to immediately execute safety protocols. Because the CPUs were blocked from being used on other processes, these remain free and so the execution of the safety procedure can start without any delay (eg. context switching).

---

[5]This is a part of context switching, which was discussed in chapter 2.4

By default most systems allow each process to use all CPUs. If the user turns off half of the CPUs of a given process and tries to create an additional workload with the library's methods for that process, he must keep in mind that the workload will also be cut in half because the threads have less processors to work on [6].

2.6   Priorities

When it comes to the priority of a process there is a big difference between a UNIX system and a windows machine. On Windows the priority is determined by the priority class of the process and its thread priority.

2.6.1   Windows

Based on the winAPI documentation[25], the classes can have the following values:

1. IDLE_PRIORITY_CLASS (0x00000040): This is the lowest priority, processes belonging to this class run only if the system is idle and can be preempted[7] by a process with a higher priority

2. BELOW_NORMAL_PRIORITY_CLASS(0x00004000): This class has a higher priority than an idle-classed process but a lower priority than a normal-classed process

3. NORMAL_PRIORITY_CLASS(0x00000080): This is the default class for all processes created by the user

4. ABOVE_NORMAL_PRIORITY_CLASS(0x00008000): This class has a higher priority than an normal-classed process but a lower priority than a high-classed process

5. HIGH_PRIORITY_CLASS(0x00000080): This class is usually used for time critical jobs

6. REALTIME_PRIORITY_CLASS(0x00000100): This is the class with the highest priority and is rarely used because it stop most of the other tasks on the calling machine

Each class can be preempted by a higher priority class besides the realtime-class. Classes categorize only processes, but not their created threads. For these the following values can be set:

1. THREAD_PRIORITY_IDLE(-15)

2. THREAD_PRIORITY_LOWEST(-2)

3. THREAD_PRIORITY_BELOW_NORMAL(-1)

4. THREAD_PRIORITY_NORMAL(0)

5. THREAD_PRIORITY_ABOVE_NORMAL(1)

6. THREAD_PRIORITY_HIGHEST(2)

7. THREAD_PRIORITY_TIME_CRITICAL(15)

These are similar to the classes mentioned above and can be interpreted alike.

---

[6]!!!BE CAREFUL, DO NOT ATTEMPT TO BLOCK A CPU IF YOU HAVE A SINGLE CORE PROCESSOR!!!
[7]If a process is preempted that means it stops executing and yields the cpu

2.6.2   Linux

On Linux however, the priority of a process is harder to be determined. This value is composed out of two main components: the nice value of the process and its thread priority.

2.6.2.1   Nice Values

Nice values can range from 20 to -19 with 20 being the nicest value and so the smallest priority and -19 being the worst value and so the highest priority. For a better understanding one could think that a process is nice, when he doesn't need the CPU and so it lets other threads to use it. In my research one thing was mentioned and that is a low nice value (hence a high priority) doesn't mean other processes won't get any CPU time. The scheduler will make them more favorable, but other processes will also get their turn for the CPU. To change the nice value of a process, in this library, I am using the calls `getpriority()` and `setpriority()` from the header `sys/resource.h`.

There is one critical thing that the caller needs to know. In order to increase the nice value of the calling process, the user can use the given methods of the library and additionally use the command `sudo setcap cap_sys_nice=ep PATH/TO/EXECUTABLE` on the built binary (the command `setcap` can change the executable to run as a privileged process, but only when called as root or with the keyword "sudo"), run it as root or build the executable program as the root-user from the beginning. You need to do this extra step, because by default any user-created processes are unprivileged.

Unprivileged processes can lower their own priority, but are not allowed to increase it more than the value of the operation *20-RLIMIT_NICE*(privileges will be discussed in detail in the next section). The RLIMIT_NICE is resource on your UNIX machine and can be set/gotten with the methods `getrlimit()` and `setrlimit()` respectively. These functions take as arguments an integer, which describes the resource we want to get or set (in this case RLIMIT_NICE) and a `struct rlimit` pointer, which describes the priority of the given resource. The structure `rlimit` has two attributes: the "rlim_cur"(also called the soft limit), that represent the current value of the process and the "rlim_max"(also called the hard limit or ceiling), which tells one user the limit of which that process can be set to. On my testing system RLIMIT_NICE is set to 13 and the limits for processes compiled by my users are zero.

```
101 int main(int argc, char* argv[])
102 {
103         struct rlimit oldcap, newcap;
104         std::cout << "getrlimit: " << getrlimit(RLIMIT_NICE, &oldcap) << std::endl;
105         std::cout << "rlimit_nice soft: " << oldcap.rlim_cur << std::endl << "rlimit_nice
    hard: " << oldcap.rlim_max << std::endl << "RLIMIT_NICE: " << RLIMIT_NICE << std::endl;
106         return 0;
107 }
```

(a) RLIMIT_NICE code

```
element@element-Inspiron-15-3567:~/Desktop/Florin/Bachelorarbeit/performance/bui
ld(main)$ ./bin/test
getrlimit: 0
rlimit_nice soft: 0
rlimit_nice hard: 0
RLIMIT_NICE: 13
```

(b) RLIMIT_NICE output

Figure 2.3: RLIMIT_NICE

Per default the process will have the nice value of 0 and this value can be increased to

the highest value allowed (-19), but cannot be decreased afterwards to a value lower than *user_nice_value = 20-RLIMIT_NICE*. A way of increasing the nice value would be to increase the RLIMIT_NICE value (also as root or with root-rights = sudo), which will allow a normal user to increase the value given until it reaches "user_nice_value". Another way would be to log in as root, use the library's functions, build the program and set the SUID as root [8].

At last you could also modify the "/etc/security/limits.conf" file and set a new max nice value for a certain user, but this is not the best solution, because that user would have the power to change priorities not only for one program, but for all its programs.

2.6.2.2   Privileges

NOTE: This section is mentioned only under Linux, because the WinAPI functions can be called without setting any other privileges

As mentioned above, when a user wants to increase the priority of a thread or process, one has to either use special commands or be signed up as root. The reason for this is the way a processor works. There are commands that use the so called *privileged* and *unprivileged* instructions.

The privileged instructions can access every corner of a processor and has no memory restrictions, such as interrupt registers, control registers or registers that define on which frequency the CPU is running. The unprivileged instructions have restrictions to internal memory segments and attributes, such as changing the default scheduler or the capabilities of a process. This way, one can restrict some applications in order to not temper with critical system areas.

To simply explain this, let's take a look at the stack of a simpler CPU such as a cortex M4 and its registers pictured in figure 2.4[9]. When a user writes a simple program such

(a) Cortex M4 Stack

(b) Cortex M4 Special Regsiters
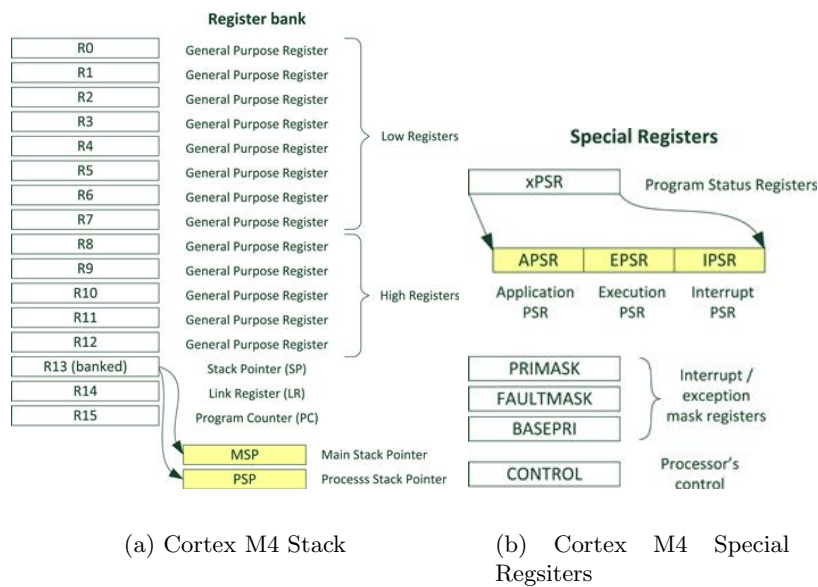
Figure 2.4: Cortex M4 register

as the addition of some variables using the ALU[10], then this can be ran in unprivileged

---

[8]The SUID is a special bit that one can set and allows normal users to run the program as they were root

[9]CPUs such as Intel or Ryzen are more complex, because they have to ensure the functionality of operating systems such as Windows or Linux, whereas Cortex processors are meant for embedded devices with simpler and smaller instruction set. However they also can work with OS such as RTOS(real time operating systems)

[10]Arithmetic logical unit

mode and so the processor will use most likely the lower registers (R0-R7) in the *Register Bank*. However if the user has a more complex program that uses for example interrupt routines from the peripheral of the CPU, which for example Intel processors constantly do for maintaining the system, then the code will have to use privileged code for setting the appropriate flags, which means the writing to registers such as the *CONTROL* register, and then land in a ISR[11]. Registers such as the CONTROL register, NVIC[12] or even the ROM table can only be accessed by privileged instructions and are protected by the MPU[13], because they have such power, that a wrong setting can shutdown the whole processor[26].

2.6.2.3 Scheduling

Unlike Windows, Linux has methods to change one's process and its threads scheduling policies.The default policy set on UNIX is called "Round-Robin Timesharing" (SCHED_OTHER). This allows jobs to be executed in a round robin fashion where each process gets an equal time-slice of a CPU. There are more than one policy which can be set. These are:

1. SCHED_OTHER

2. SCHED_BATCH

3. SCHED_IDLE

4. SCHED_FIFO(real time policy)

5. SCHED_RR(real time policy)

The difference between SCHED_RR (normal Round Robin scheduling) and "Round Robin Timeshare"(SCHED_OTHER) is that the realtime policy lets us to coordinate the priorities for that schedueling policy's queue.
SCHED_BATCH and SCHED_IDLE are two normal prioritised policies, which differ from SCHED_OTHER, but not enough for me to focus too much on them. The only differences are that BATCH schedules a process less frequently, if the it gets the CPU very often and IDLE is the equivalent to a process with a nice value of 19 (very nice process $<=>$ lowest value).
You can get the current policy of your process by calling `int sched_getscheduler(pid_t pid)` from the `<sched.h>` header file.
Each of the policies mentioned above has a range of priorities levels, which can be get using the `sched_get_priority_max(int policy)` and `sched_get_priority_min(int policy)` methods found in `<sched.h>`[14].
On my Linux Notebook I have the following values shown in figure 2.5.
You can only change the priority of real-time policies, hence when you set the policy of a thread to OTHER, IDLE or BATCH you can't change the priority of that thread of execution.
As you can observe in figure 2.5 only two of these have a "real" range. SCHED_RR and SCHED_FIFO are characterized as real-time policies and have a higher priority than the others. When two processes with SCHED_RR and respectively SCHED_FIFO have to share a CPU, ironically the one that was placed first in that CPU's queue will get to run its job. Both of these policies can lose access of their CPU, if they finish execution, call sched_yield() or a syscall is called and a higher priority process preempts them (a process with a lower nice value appears in the queue or the user changes the value himself). SCHED_RR can also lose its access if the timeslice if the job expires.

---

[11]Interrupt Service Routine
[12]Nested Interrupt Vector: a table containing all of the system's interrupts
[13]Memory protection unit
[14]These can be different depending on the calling xUNIX machine

```
std::cout << "sched_get_prio_min fifo: " << sched_get_priority_min(SCHED_FIFO) << std::endl;
std::cout << "sched_get_prio_max fifo: " << sched_get_priority_max(SCHED_FIFO) << std::endl;
std::cout << "sched_get_prio_min rr: " << sched_get_priority_min(SCHED_RR) << std::endl;
std::cout << "sched_get_prio_max rr: " << sched_get_priority_max(SCHED_RR) << std::endl;
std::cout << "sched_get_prio_min other: " << sched_get_priority_min(SCHED_OTHER) << std::endl;
std::cout << "sched_get_prio_max other: " << sched_get_priority_max(SCHED_OTHER) << std::endl;
std::cout << "sched_get_prio_min idle: " << sched_get_priority_min(SCHED_IDLE) << std::endl;
std::cout << "sched_get_prio_max idle: " << sched_get_priority_max(SCHED_IDLE) << std::endl;
std::cout << "sched_get_prio_min batch: " << sched_get_priority_min(SCHED_BATCH) << std::endl;
std::cout << "sched_get_prio_max batch: " << sched_get_priority_max(SCHED_BATCH) << std::endl;
```

(a) sched_prio_range code

```
[100%] Built target test
element@element-Inspiron-15-3567:~/Desktop/Florin/Bachelorarbeit/performance/build(main)$ ./bin/test
sched_get_prio_min fifo: 1
sched_get_prio_max fifo: 99
sched_get_prio_min rr: 1
sched_get_prio_max rr: 99
sched_get_prio_min other: 0
sched_get_prio_max other: 0
sched_get_prio_min idle: 0
sched_get_prio_max idle: 0
sched_get_prio_min batch: 0
sched_get_prio_max batch: 0
```

(b) sched_prio_range output

Figure 2.5: sched_prio_range

2.7 Synchronization

When working with threads, the programmer cannot forget that these share the same stack, therefore they share the same variables. When two or more threads start their execution (eg. a simple addition), one cannot tell when will they perform what (if their priorities weren't tampered with).

There are two main operations threads can perform on a variable: `read()` and `write()`. Even a simple addition needs to first read the value from a register, add a number to it and then write the new value back to the register. Reading from a variable is not problematic, because its content will always stay the same, but writing to one is a whole different story. Let's take a look at the following code:
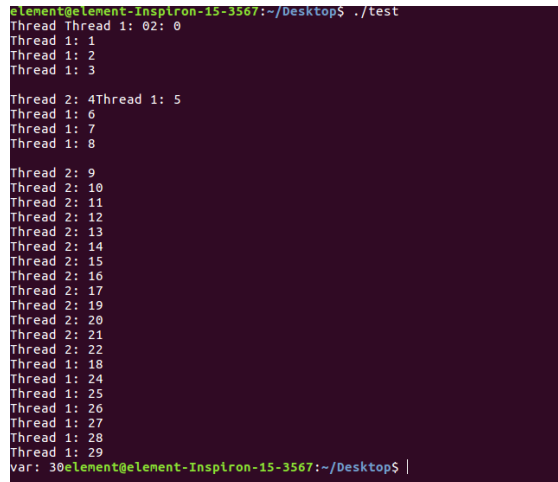
```
1 #include <iostream>
2 #include <thread>
3
4 void addition(int& var, int thread)
5 {
6     for(int i=0; i<10;i++)
7     {
8         std::cout << "Thread " << thread << ": " << var << std::endl;
9         var++;
10     }
11 }
12
13 int main()
14 {
15     int var = 0;
16     std::thread t1(addition, std::ref(var),1);
17     std::thread t2(addition, std::ref(var),2);
18     t1.join();
19     t2.join();
20     std::cout << "var: " << var;
21     return 0;
22 }
```

Figure 2.6: Thread's behavior without synchronization - Source code

Here we create two threads with the sole purpose of adding a common variable. Most people would expect, because of the order of creation, for t1 to do his job and afterwards t2, but this is not the case.

As you can observe in figure 2.7, the prints make no sense. The threads do not follow a sequentially pattern so the variable can be increased by `t1` for a time then by `t2` and for the rest of the remaining time by `t1` again.

Something that cannot be seen in the output would be the scenario when `t1` and `t2` try to access the variable and increment it at the same time. No one can accurately predict what the result would be (this is also known as *Unexpected behavior*). To resolve this problem in classic C people came up with the idea of a locking mechanism called `Mutex`.

Figure 2.7: Thread's behavior without synchronization - Output

2.7.1   Mutex

Mutexes are guards that can block other threads the access to variables inside a user-defined block of code (also known as *scope*). These allow the thread that called `lock()` exclusive access to the variables inside that scope until the thread calls `unlock()`[23]. If another thread tries to lock the mutex for himself while the mutex is being used, then it will land in a waiting state until that lock is released. The accessing order of the variable is "First come, first served".

But as many implementations, this method can create problems, especially if the programmer forgets to `unlock()` the mutex. If this ever happens, it can lead to other threads being stucked in a waiting state forever (this is also called a *Deadlock*)[27].

This method requires a lot of concentration from the programmer and a good overview of all locking and release point. Fortunately the *Standard C++ Library* implemented new ways to use mutexes like *lock_guards*, which unlock themselves when the user goes out of scope, *unique_locks*, *shared_locks* and even *timed_mutexes*. Additionally they made a more simplistic technology for threads to access only common variable (and not a whole scope) called *Atomic Variables*.

2.7.2   Atomic Variables

Atomic variables can be seen as wrappers for primitive data types, like `bool`, `int`, `double`, `float`, etc., but also for defined structs like `uint32_t`, `uintptr_t` or `int_fast32_t`

Basic syntax of this wrapper is: *atomic<data_type/class>* or (if defined) *atomic_data_type* and it will grant the calling thread exclusive access to a certain variable while reading from or writing to it.

To read from an atomic variable one could just use the "="-operator or (preferred) the `atomic_var.load()` method. Writing to an atomic variable should only be done by using the `atomic_var.store()` method, because this way you can expect that no one else besides the calling thread is accessing that variable[28, 29].

One must be very careful when using atomic operations on a variable. To explain this in detail let's observe the following example:

A simple incrementation of a variable:

1. x++

2. x += 1

3. x = x+1

All of these are atomic operation, but number three is different from the first two. The first and second operation are both *atomic increments*. The third however, needs to apply more operations on the variable. Here there is an *atomic read*, to get the value from x and store it in a register, afterwards we add one to it and there is an *atomic write*, to set the variable to its new value. If you are working with multiple threads any other thread can come in between the `write` and the `read` and temper with the value, because there is no exclusivity while the value is stored in the register. The atomic library comes with additional member functions to make sequences of reading and writing easier and specialized member function that define bitwise operations on a variable, but they go way too deep in complexity and understanding for me to focus too much on them[29].

2.8   CMake

Despite the numerous implementations for the C++ language, there are still some hardships that need to be overcome when working with it. When it came to building, many people used os-dependent technologies such as the *GNU Compile Collection*(short GCC) for Linux[30] or MingW, which supports the usage of the GCC compiler, on Windows[31]. Despite the existing methods, developing for multiple architectures was always complicated as code needed to be moved from one computer to the other and recompile, only to see that some incompatibilities still existed between the code's lines, because of small differences between compilers. The demand "for a cross-platform build environment" was so high, that a new tool called *CMake* emerged.
"CMake is an extensible open-source system that manages the build process in an operating system and in compiler-independent manner". As mentioned on their website, the system is used with native build environments and can be used to build binaries of a specific projects. These binaries can be executable files or compiled libraries such as *.o-files (also known as Object-files) on Linux or *.dll* (Dynamical Linked Library) on Windows. In order to use this program one has to first write basic text files and call them *CMakeLists.txt*. The content of these files needs to be written in an interpreted language. Its Syntax goes as follows: `COMMAND(args...)`. Each subdirectory of the project needs to contain a *CMakeLists.txt* file that will later be linked through a command to the main *CMakeLists.txt* file[32].

2.9   Need for action

Although the presented technology uses the C and C++ syntax and logic, there is still no universal tool to cover the most relevant operating systems out there when it comes to the machine's performance. This is where this library comes in handy.
The goals are:

1. summarize the complexity between operating systems into easy-to-understand methods

2. supply the user with numerous operations to ensure homogeneity and flexibility usage in order to cover most user-cases in the industry

3. encourage in-house development of product based test and benchmarks

3. Library Overview

This chapter will cover the library's build system and its main components, the system and workload components, as well as their logic.

The library has two main components. The first one is the workload, which can vary depending of the user's input. This will create a system specific amount of threads and make them run a simulation function. The number of threads will stay constant and the time for running the workload task will be set accordingly for each input.

The second components is the system. This comes with functions, which can easily change a thread's priority, a system's scheduling policy and deliver statistics of the user's computer.

Basic operations are implemented to work on most operating systems, but there are some exceptions because of the differences between OS implementations, which makes them independent from one another.
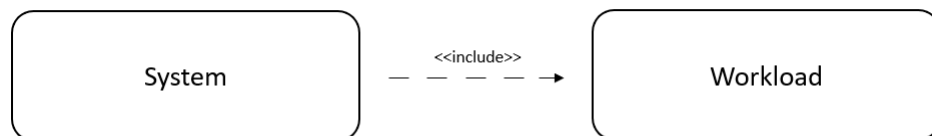
Figure 3.1: Library's components

3.1   Workload

This part of the library is mainly a C++ class, that lets the user create a specific workload.
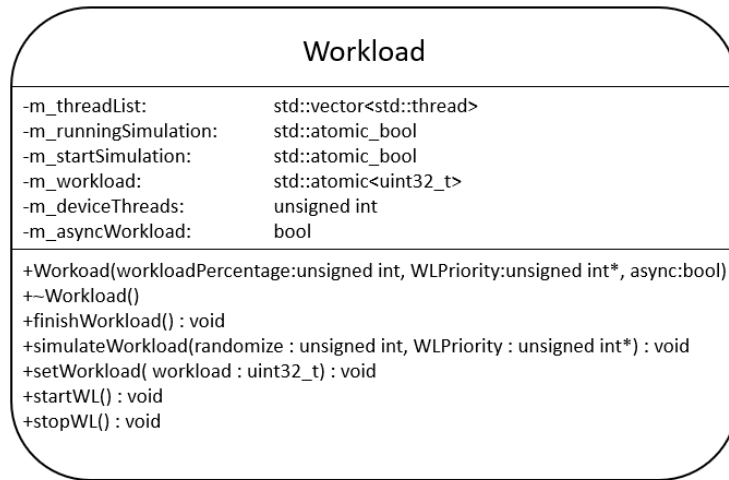
Figure 3.2: Workload class UML

3.1.1   Attributes

The class has six private Attributes:

1. `threadList`: is a list of type *std::vector*, that holds the current number of working threads after their initialization

2. `runningSimulation`: is a flag that tells the user the status of his working threads. This will be set to true once at least one thread in *threadList* has started executing a routine

3. `startSimulation`: is a switch that starts a simulation function (in our case *simulateWorkload()*). This is in order to synchronize threads and start them as simultaneously as possible

4. `workload`: hold the current workload set by the user. This is meant for future implementations in order to let the user change workloads dynamically at runtime

5. `deviceThreads`: is the number of threads the current machine can handle at the same time. This depends on the processor that the executable is running on, therefore can differ on different device

6. `asyncWorkload`: this is a flag for testing purposes in order to simulate a "real" workload. This will be explained in detail later in this chapter.

The names start with "m_" in order to tell the programmer that the variable is a member of the class. This makes the code more readable for the programmer and informs him about the variable's purpose at the same time, without the need to check the class's definition.

3.1.2 Methods

The class's methods are:

1. `Workload()`: this is the class's constructor and takes three arguemnts:

   (a) workloadPercentage: workload percentage set by the user to simulate

   (b) WLPriority: this is a pointer to a OS-specific thread's priority. If this is set to *NULL*, all threads will have the same (default) priority

   (c) async: lets the user to create a more "real" workload.

   The first thing the constructor checks when a new object is created is the validity of the passed workload, which cannot be higher than 100 or lower than zero. Next it sets the *runningSimulation* and *startSimulation* attributes to false for a synchronized start. Afterwards it measures the current workload for ten seconds (the time for the measurement can be set to less or more, but ten seconds is in my opinion a proper chunk of time) in order to warn the user for a possible system overload. At last, the workload's given percentage is stored in the attribute *workload*, an output will inform the user about the number of threads that will be created and a loop creates the threads, which will run class's *simulateWorkload()* method.
   The created threads are pushed into an *std::vector* using *emplace_back()*. I opted for this method instead of *push_back()*, because the compiler doesn't have to create the thread, store it temporarily and push it afterwards, but it uses the *std::move* mechanism, where the object is moved directly after the creation. This way we spare ourselves some extra memory (which is a very important factor, if this is used on an ARM architecture)

2. `~Workload()`: this is the class's deconstructor. It is set to default and should be changed for future development. For the moment, I used the *defaul* keyword, but (at some point) the purpose of the destructor should be the object's clean up. For example the joining of the threads in the *threadList* attribute.

3. `finishWorkload()`: this function iterates through the class's thread list, checks if these are joinable (in case some user used detach for some reason) and joins them. In case a thread cannot be joined the *TID* will be printed to the console

4. `simulateWorkload()`: this takes two arguments

   (a) randomize: this is set the index in the thread's construction loop, but should be changed to something better in the future

   (b) WLPriority: is the priority passed in the constructor. If this is not `NULL`, the given priority will be set

   First the functions increases the priority of the calling thread to maximum, then it waits for all threads to be created. When that happens the attribute *startSimulation* will be set to true. This way we ensure a finer synchronisation between the threads, because reading from a variable and leaving the loop should take less time than the time gap between the thread's creation. Next, each thread sleeps for a couple of milliseconds, if *asyncWorkload* is set so that each thread starts with a small delay in order for the library's workload to be above zero at all times. One can imagine the workload as a sinus curve if the flag is set and a square wave function otherwise (as seen in Figure 3.3). Afterwards the actual workload loop will start. I decided to recalculate the sleep time based on the current workload here, in order to support future dynamically changing workload for the simulation. The frequency of the
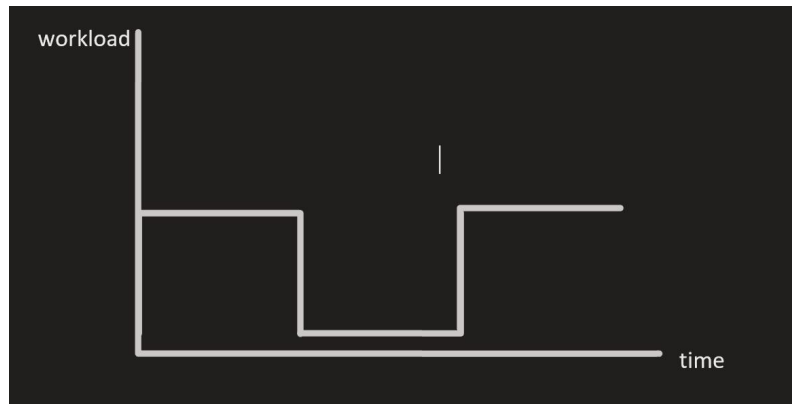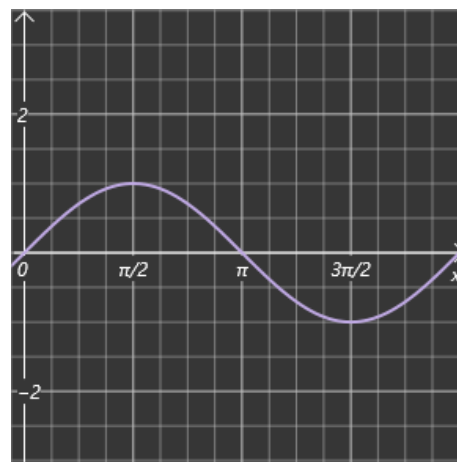
(a) `asyncWorkload` not set

(b) `asyncWorkload` set

Figure 3.3: Workload Time Diagrams

workload is hard coded to $(\frac{2*workload}{100})$Hz,because a cycle of working and sleeping for a thread is half a second and the system measures the workload once a second so this highly depends on how often the library's system component makes its time measurements.

5. `setWorkload()`: this method's concept is to let the user dynamically change the thread's priority

6. `startWL()`: checks if all threads that needed to be created were saved in the thread list. If this is the case it sets *simulationStart* and *runningSimulation* to true

7. `stopWL()`: sets *runningSimulation* to false. It makes sense to call this method before *finishWorkload()*, in order to properly stop all workload threads

3.2   System

This part of the library mainly contains methods to modify a thread's or process priority, the number of CPUs the process can run on (mentioned in the cpu's affinity section 2.5)and the scheduler's policy (mentioned in the operating system's scheduling policies 2.6.2.3).


```
                          System Methods

# WIN32
setCPUAvailability(proc: HANDLE&, mask: uint64_t): int
checkCPUAvailability(proc: HANDLE&): int
increaseSchedClass(): int
decreaseSchedClass(): int
increaseThreadPrio(id=0 : HANDLE): int
decreaseThreadPrio(id=0 : HANDLE): int

# LINUX
readCPUTime(cpu: int): std::string
returnData(cpu : int): std::vector<uint64_t>
printPolicy(policy : int): void
changePolicy(policy : int, pid : int): int
decreaseProcessNiceValue( pid=0 : int): int
increaseProcessNiceValue( pid=0 : int): int
decreaseThreadPrio( pid=0 : int): int
increaseThreadPrio( pid=0 : int): int

# NON-OS-SPECIFIC
getSpecificTime(which : int): uint64_t
convertTime(timeToConvert : int): uint64_t
writeOverallStats(stat: double, statName: std::string): int
getProcessTimes(kernel: uint64_t&, user: uint64_t&): int
getCPUTimes(kernel: uint64_t&, user: uint64_t&, idle: uint64_t&): int
calculateAndShowLoad(duration: double, processWLList: std::vector<double>&, systemWLList: std::vector<double>&): int
calculateSystemLoad(duration: int): double
```

Figure 3.4: System Methods

3.2.1   Defines

1. `STR_ERR`: macro returned by a function if it failed

2. `IDLE_TIME`: macro that represents idle time of the system

3. `USER_TIME`: macro that represents user time of the system

4. `KERNEL_TIME`: macro that represents kernel time of the system

5. `schedPrioList`: vector containing scheduler classes for a WINDOWS machine in ascending order(mentioned in 2.6.1)

6. `threadPrioList`: vector containing thread priorities for a WINDOWS machine in ascending order(see 2.6.1)

7. `sched_attr`: struct containing thread's attributes for a LINUX machine[33]

3.2.2   Methods

1. Windows:

   (a) `mergeFILETIME()`: takes a FILETIME structure as argument, merges the two 32-Bits attributes of the struct into a uint64_t variable and returns it

   (b) `setCPUAvailability()`:

       i. proc: is a windows identification struct for processes of type `HANDLE`
       ii. mask: a 64-Bit value representing the cpu affinity for most computers out there. Each bit represents a cpu. If a bit is set to 1,that means the process can run on the CPU corresponding to that bit.

(c) `increaseSchedClass()`:this method gets the current scheduler class and checks if it is already set to the maximum value allowed. In that case it will not do anything, throw an error and return 1. Otherwise it will get the index of the current class in the *schedPrioList*, increase the index by one and set it accordingly. This allows the user to test its program step by step without knowing the actual values in the list. If the user wants to set a certain value for his thread, he can simply use the *winAPI* functions. If one needs to increment the class to the maximum value possible, one can use a while loop in a catch and try block.

(d) `decreaseSchedClass()`: does almost the same as *increaseSchedClass()*, but instead of increasing the scheduler class's priority, it decreases it. If one wants to decrease the class's value to the minimum one can use a while loop in a catch and try block.

2. Linux/Unix:

(a) `readCPUTime()`: this functions takes an argument of type integer and reads data about the CPU specified by the given number using the function *getline()*. For example if your machine has four CPUs, then passing a one will deliver data from the first CPU, passing a two will deliver data for the second CPU and so on. Passing a zero is also acceptable and will deliver data for the whole system (instead of getting individual segments for each CPU, the delivered segments will contain the sum of all CPUs for that specific segment). The data read comes from the "/proc/stat" file. As described in the manual page, the proc filesystem provides informations about the kernel structure or in other words about all processes on the calling system. The "/proc/stat" file contains statistics about the system.

This component uses only the first line and the lines which begin with the name *cpuN*, where *N* stands for the CPU specified by the argument passed to this function. Each line that contains statistics about the CPU has the following pattern[34]: The most important statistics for the library are[34]:

name	user	nice	system	idle	iowait	irq	softirq	steal	guest	guest_niced

Figure 3.5: Library's components

i. name: is either cpu or cpuN, where cpu stands for the whole system and cpuN for a cpu specified by N, where N$\in$ [0;max_number_of_CPUs]

ii. user: is the time spent in user mode

iii. nice: is the time spent in user mode with low nice values

iv. system: is the time spent in system mode

v. idle: is the time spent for idle tasks

If the function is successful, it always returns an `std::string` containing the line specified by the passed argument, else it return the macro `STR_ERR`.

(b) `returnData()`: this function takes an integer as its argument specifying the cpu that we want data from. If the integer is zero, the function returns data from the whole system, one for the first cpu, two for the second cpu and so on. If the given integer is higher than the value returned by *std::thread::hardware_concurrency()* or lower than zero the function will fail and throw an exception, else the argument will be passed to the *readCPUTime()* method, parse the string returned by it and return the data as an *std::vector* containing *uint64_t* values.

(c) `printPolicy()`: this is a utility function that takes an integer as the argument and prints the policy corresponding to it. Possible values are for the arguments are specified in section 2.6.2.3.

(d) `changePolicy()`: this functions takes two integer as its arguments:

    i. policy(int): new policy that should be set

    ii. pid(int): the process's *PID*. If this is zero this function will change the policy of the calling process

First this function uses system calls, *SYS_sched_getattr()* to get the current *sched_attr struct* specified in the linux man pages and *SYS_sched_setattr* to set the new one for the given process specified by the passed arguemnts[33].

(e) `decreaseProcessNiceValue()`: this method takes an integer as its argument specifying the process's PID and is similar to *decreaseSchedClass()*, but instead of checking a local vector list, it uses *getpriority()* with *PRIO_PROCESS* and *PID* to get the current nice value, saves it temporarily and decreases it by one. Then it uses *setpriority()* to set the new value with the same arguments passed to *getpriority()*. If the nice value reaches 20, the function will throw an exception. The method will fail only if *set/getpriority()* fails.

(f) `increaseProcessNiceValues()`: this method does the same as its counterpart function mentioned above, but instead of decreasing the value, this method increases it.

3. NON-OS-Specific:
NON-OS-Specific functions refrain to methods that either don't depend on any of the OS-Specific libraries such as the UNIX header files and the winAPI or they give the impression of identical behaviour, while in reality each case is manged with a lock-guard.

(a) `getCPUTimes()`: this function takes the following arguments

    i. kernel(uint64_t reference[1]): kernel time -> time the CPU spent for more "privileged" methods like system calls or interrupts

    ii. user(uint64_t reference): user time -> this refers to any other methods for example a simple loop written by the user

    iii. idle(uint64_t reference): idle time -> this time shows how long the system has been idle

This function gets the three times specified by the arguments and writes them in the passed references for further usage.
On Linux it uses the *returnData()* method with zero as its argument and it saves the first four entries of the returned vector. The zeroth and first entry sum up the user time, the second is the kernel time and the third is the idle time.
On Windows it first creates three FILETIME variables and uses *GetSystem-Times()* to get the three specified times. If *GetSystemTimes()* fails, an exception will be thrown and the function will return one, otherwise it will call *mergeFILETIME()* (because a FILETIME variable is a struct that contains two 32-bit attributes and one needs to concatenate the lower bits to the upper bits to get the real time as a 64-bit variable).

(b) `getProcessTimes()`: this function takes two uint64_t references as its arguments and writes the user and kernel times in them. The methods used to get these times do not deliver the idle time (logically thinking there is no user case

---

[1]A reference is like a pointer but the variable passed cannot be *NULL*

when this might be important).

On Linux the function calls the *times()* method specified in the *<sys/times.h>* header file. This writes the required times of the calling process in two attributes of type *clock_t* called *tms_utime* and *tms_stime* which belong to a data type called *struct tms* (this also contains the kernel and user time of any process children created by the current process, but these are irrelevant for this implementation as I use threads instead of processes).

On Windows it uses *GetProcessTimes()* to get the times similar to *getCPUTimes()* and uses *GetCurrentProcess()* to specify the current process. *GetProcessTimes()* also delivers the creation and exit time, however these are irrelevant for this implementation. If *GetProcessTimes()* fails the function will throw an exception and will return one.

(c) `convertTime()`: this is a utility method and takes a uint64_t as its argument representing the time.

On Linux *sysconf(_SC_CLK_TCK)* is called, which returns the system's clock ticks per second value. If this fails, an exception is thrown and the functions returns minus one, otherwise it return the seeked time in milliseconds.

On Windows it returns the given argument multiplied by 0.1, which also represents the time in milliseconds.

(d) `getSpecificCPUTime()`: the method is more utility based and similar to *getCPUTimes()*, but instead of writing its times into the passed arguments, it returns only one time described by the given parameter. Its type is an integer and describes which CPU time should be returned as a uint64_t value. Possible arguments are *IDLE_TIME*, *USER_TIME* and *KERNEL_TIME*, which are defined in *system/system.hpp* (also mentioned above in the DEFINES section) file and extend to zero, one and two respectively.

(e) `increase-/decreaseThreadPrio()`: this is similar to *increase-/decreaseProcessNiceValue()* or *increase-/decreaseSchedClass()*.

On Linux the method takes an integer as parameter called *id*, which it gets the current TID's priority, if *id* is zero (which is set by default) or the priority specified by *id* with the help of *sched_getsched()*. Afterwards it initializes a *sched_attr* struct, increases/decreases the priority (throws an error if the maximal/minimal priority is reached) and sets the priority through a system call[2].

On Windows it does the same. The only difference is in the parameter and the functions called to get and set the thread's priority. Here the *GetThreadPriority()* and *SetThreadPriority()* are used which expects a HANDLE type as their arguments. That's why we also pass a HANDLE to this function instead on an integer as TID.

(f) `calculateSystemLoad()`: the function calculates and returns the system's current workload as a double value[3]. It takes an integer as its arguments, which specifies the duration of the check. On Linux it adds the kernel time to the user time, divides it to the addition of kernel, idle and system time and multiplies the result with 100 to get a workload as a percentage.

$$workload = \frac{\text{t}_{user} + \text{t}_{kernel}}{\text{t}_{user} + \text{t}_{kernel} + \text{t}_{idle}} * 100 \tag{3.1}$$

On Windows however kernel time also contains the idle time of the system (as

---

[2]Using the system call is preferred, because it supports future development of the function for tempering with more thread's attributes and not only the their priority

[3]This is mainly used in the workload class to tell the user the system's current workload and not overload the system

specified in the winAPI documentation), so in order to get a meaningful and correct result we have to subtract idle time from kernel time.

$$workload = \frac{\text{t}_{kernel} - \text{t}_{idle} + \text{t}_{user}}{\text{t}_{user} + \text{t}_{kernel}} * 100 \tag{3.2}$$

(g) `calculateAndShowLoad()`: this functions takes three parameters:

   i. duration: the duration of the check while the load simulation is active
   ii. processWLList: a *std::vector* that contains entries with the process's workload in one seconds intervals. So the number of entries is also defined by the duration of the check
   iii. systenWLList: also a *std::vector*, which contains the system's workloads during the check. It size also depends on the duration of the check

It also calculates the system's and process's workloads using the previous mentioned method. But instead of just printing it to the screen it saves the values into lists (*std::vectors*) used later for writing logging files. Before terminating itself, it also determines the time which the system was not idle using the *convertTime()* and *getSpecficCPUTime()*

(h) `writeOverallStats()`: this operations takes two arguments:
1) *stat*, which represents the number of the statistic that will be written to the log file
2) *statName*, a string representing the name of the stat
After calling this functions, it will try to create and open a *comma separated file*(CSV) named "overall_stats.csv". If this fails an exception will be thrown and the function returns -1. Otherwise it will write the given stat with the given name string passed by *statName* and close the file.

(i) `wirteRuntimeStats()`: similar to the latter, this functions writes all statistics given by its first argument with the name passed into the second argument to a CSV file. If the CSV file could be successfully created and opened, it will write the statistics and close if afterwards. Otherwise the method will throw an exception and return -1.

3.3 Build System - CMAKE

As described in 1.4.2 the goal of this thesis is to create methods supporting the two big operating systems in the industry: Linux and Windows. In order to do that, the library comes with additional support for projects that use CMake for their build system. When calling the *cmake* command on the main *CMakeLists.txt*, the system will generate different files depending on the calling OS. When called on Linux, it will generate *Make* files that can be used with the make mechanism that comes with most (almost all) *UNIX* distributions to build the files specified in *CMakeLists.txt* of each directory. When called on Windows, the program will directly create the binaries specified in the main CMakeFileLists.txt. If the Windows machine has *Visual Studio Studio* installed, CMake will also create a solution of the project that can be used for further development and debugging. To create the main executable one would need to use the solution and build the executable from the IDE[4]. Alternatively one could directly use the compiler from the command line (like the MSVC compiler - cl.exe) and run the executable afterwards.

Another thing that should be mentioned is that CMake syntax enables the user to link the components to the library statically or dynamically[5].

3.3.1 Implementation

As already mentioned, when using *CMake* each directory needs a *CMakeLists.txt*. The main file is the most important file, because it sets the requirements for the whole project and also binds all of the other directories into a single top-level-point.

The first requirement for this project is the minimum cmake version, which is set from 3.12 to 3.21. Then a failing criteria is set, for the user to know, that the library doesn't support older versions than 3.12. This way the user does not have to have the newest versions, but we should also make them update their CMake program if it's too old. Afterwards some CMake standard variables are set to make the user have at least C++17 enabled. These settings are written for two main reasons:

First, the library uses newer elements for its components, therefore older versions won't be compatible and so errors will occur.

Second, the library should encourage the users to start implementing modern and

```
PERFORMANCE
|    .gitignore
|    CMakeLists.txt
|    LICENSE
|    README.md
|
+---scripts
|        evaluation.py
|
+---source
|        CMakeLists.txt
|        test.cpp
|
+---system
|        CMakeLists.txt
|        system.cpp
|        system.hpp
|
\---workload
         CMakeLists.txt
         workload.cpp
         workload.hpp
```

Figure 3.6: Library's directory tree

up-to-date standards and companies should opt for a the newer tools as they often have more to offer(in my opinion C++17 cannot be even seen as a new standard anymore, considering that at the time of writing the newest, with a final version, Cpp Standard is C++20 and with C++23 already in development with full integration for the GCC compiler).

---

[4]Integrated Development Envoirement, such as Visual Studio 2019, InteliJ or PyCharm

[5]Static Library means that the linked libraries will be part of the executable, while the code from a Dynamically Linked Library will be linked at runtime, thus resulting in a much smaller sized executable. A good hint for the user when trying to link this library

After that the library prints a message to tell the user what operating system was recognized by CMake and then we define the name of the project, the version, description and its programming language.

Finally we tell CMake which directories contain the code for our project by adding them to the include search path and linking the targets that are defined in those directories to the main executable target created in the *CMakeLists.txt* that can be found in the source subdirectory[6].

Unfortunately, during the development phase, linking the components dynamically lead to errors and because of the lack of time, the components were linked statically.

Each component has its own *CMakeLists.txt*. The workload components first creates a static library/target with the appropriate header and source files, sets the *THREADS_PREFER_PTHREAD_FLAG* to ON and searches for the *pthread* library. If this is found the compilation continues, otherwise an error will be raised, because of the component's thread implementation. Afterwards it includes the system's component directory to its search path and links against it and against the pthread library. At the end, it specifies that the compiled binary (windows .lib-file and Linux .o-file[7]) needs to be created in the binary directory specified by *CMake* in a subdirectory called "lib".

The system component has a much simpler *CMakeLists.txt*. It only creates the target with the specific header and source files and specifies where the compiled library will be created, which is the same directory as for the workload component.

A detailed directory tree of the library can be seen in figure 3.6

---

[6]When compiling with CMake it is a good practice to create an extra build directory, and compile the project from that directory, because all of the compiled files will be generated in the calling directory

[7]If the library was dynamically linked on windows we would get a .dll-file and on Linux a .so-file

4. Accuracy testing

In order to show the accuracy of the methods and establish a meaningful use of (some of) the library's components, the following method was created.
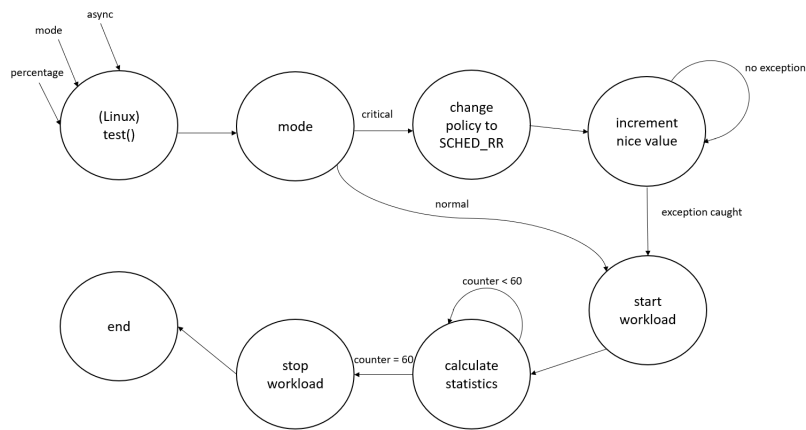
Figure 4.1: Testing method - Linux
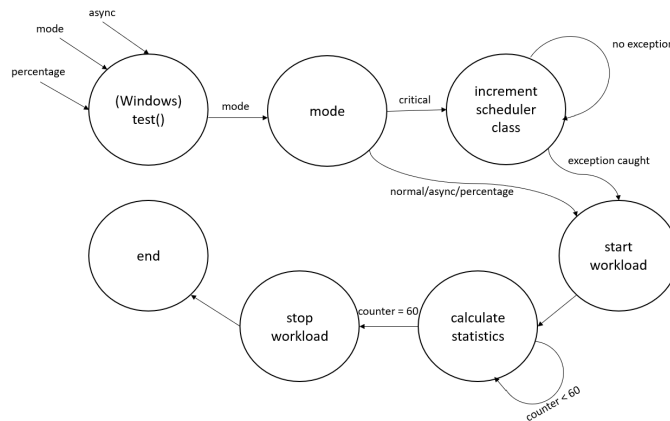
Figure 4.2: Testing method - Windows

Here we pass the most crucial parameters:

1. percentage: in order to specify how big the workload created should be

2. async: to verify if the asynchronous mode makes a huge impact on the system

3. mode: which can be set either to CRITICAL (define that extends to one) or *NORMAL* (define that extends to zero).

The method first defines two lists that will contain the system's and process's workloads. Additionally it sets two flags to true to increase the process's priority (Linux: nice value / Windows: sched class) to maximum and a variable called *prio* to null, which will later be set to *MAX_PRIO* (defined in *workload.hpp*) to increase the priority of the workload's threads in case the mode was set to critical. The priority's incrementation is called in a *try-catch* block. If this catches an exception we shall set the respective flags to false in order to avoid an endless loop. Afterwards we create an object of type *Workload* and we pass it the percentage, variable prio's reference and the asny argument. Then we start the workload and call `calculateAndShowLoad()` in order to see and get the system's statistics. Then we call `Workload.stopWL()` to stop the loops that create the artificial workload and `Workload.finishWorkload()` in order to also correctly terminate(join) the started workload threads. At last we use the `writeRuntimeStats()` function to write the statistics from the lists that we initialised at the begging of the test. This order is necessary for the user to get meaningful results. Small tests in the developing phase of this library showed that computers can't handle well a huge load and IO-operations at the same time. That's why the writing of two or more statistics needs to happen only after the workload was finished.

The method can be called as followed:

```
108 int main(int argc, char* argv[])
109 {
110         //test(CRITICAL, 10, true);
111         test(NORMAL, 90, false);
112
113         return 0;
114 }
```

Figure 4.3: Main testing method

To run the experiment the following steps were executed:

This will create two log files:*overall_stats.csv* and *runtime_statistics.csv*.

4.1 Results

4.1.1 Linux

The Linux machine under testing has the system's attributes and OS:

1. OS: Ubuntu 16.04

2. Processor: Intel Core i5-7200U, 2.50GHz

3. Cores: 4

4. Threads per Core: 2

5. Architecture: x86_64

4.1.1.1 Normal Mode

Normal mode was tested with a 90% workload. The results were as expected. Additionally the following statistics were logged in the *overall_stats.csv* file.

1. Average System Workload: 90.2126

2. Average Process Workload: 89.5858

3. NIT[1](time process was working out of 60 seconds): 60.00s

Figure 4.4: Library comparison with Linux's top-command: normal mode
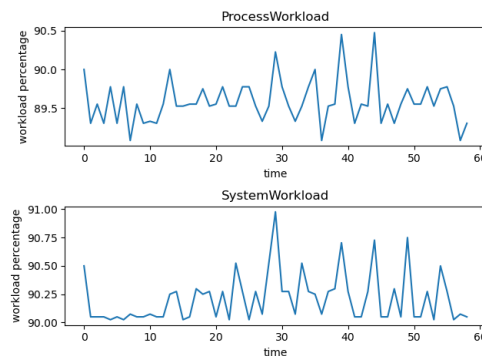
Figure 4.5: Overtime statistics: normal mode

---

[1]NotIdleTime

4.1.1.2   Critical Mode

Critical mode was tested with a 10% workload. The results were as expected. Additionally the following statistics were logged in the *overall_stats.csv* file.

1. Average System Workload: 10.564

2. Average Process Workload: 9.97427

3. NIT(time process was working out of 60 seconds): 59.799s

Figure 4.6: Library comparison with Linux's top-command: normal mode
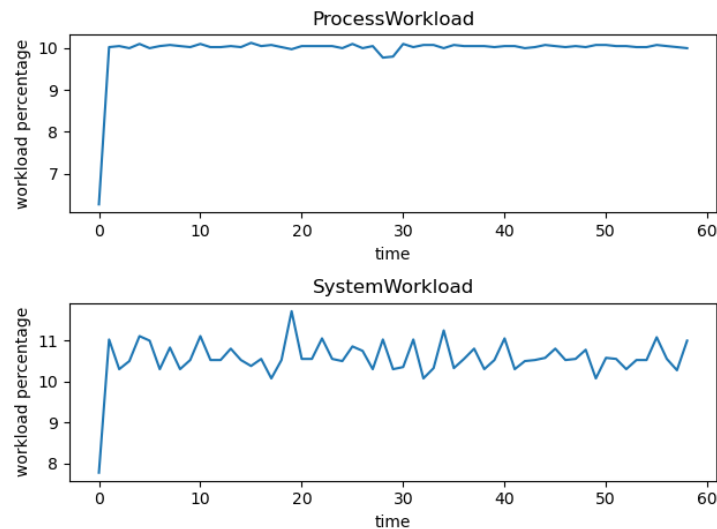
Figure 4.7: Overtime statistics: normal mode

4.1.2 Windows
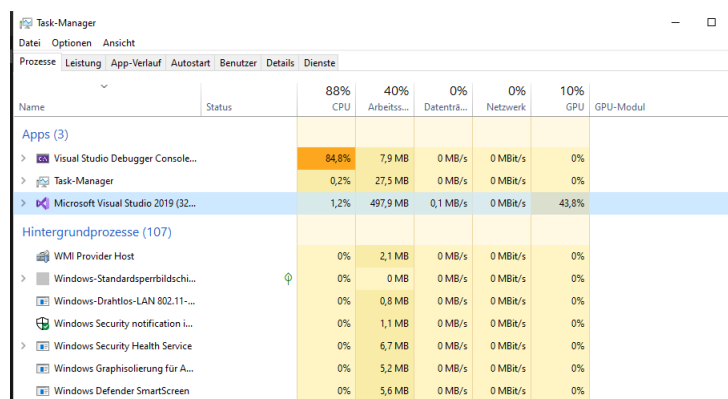
The Windows machine under testing has the following system attributes and OS:

1. OS: Microsoft Windows 10 Home

2. Processor: Intel Core i7-8700, 3.20GHz

3. Cores: 6

4. Logical Cores: 12

5. Architecture: x64-based-processor

4.1.2.1 Normal Mode

As with Linux, normal mode was tested with a 90% percent workload. The following values were logged in the comma separated files:

1. Average System Workload: 87.8445

2. Average Process Workload: 85.3584

3. NIT: 66.040s

Figure 4.8: Library comparison with Windows's task manager: normal mode
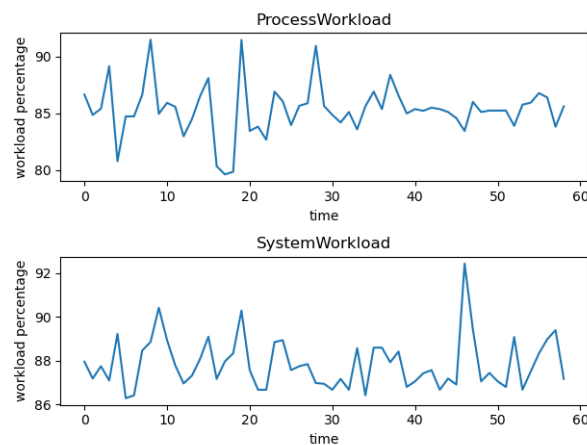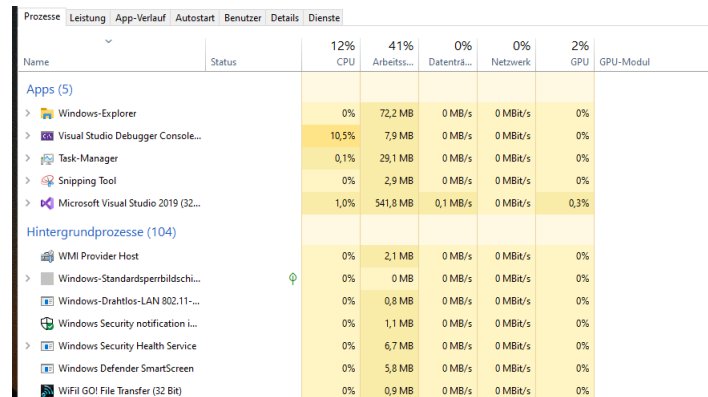
Figure 4.9: Overtime statistics: normal mode

4.1.2.2   Critical Mode

Critical mode was tested with a 10% workload. The following values were logged in the comma separated files:

1. Average System Workload: 10.351

2. Average Process Workload: 8.22164

3. NIT: 60.907s

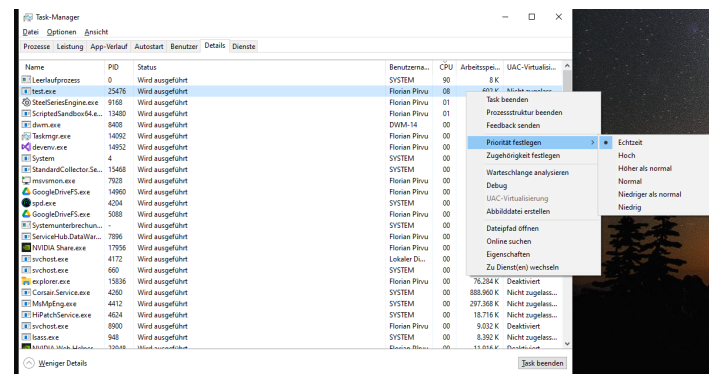Figure 4.10: Library comparison with Windows's task manager: critical mode

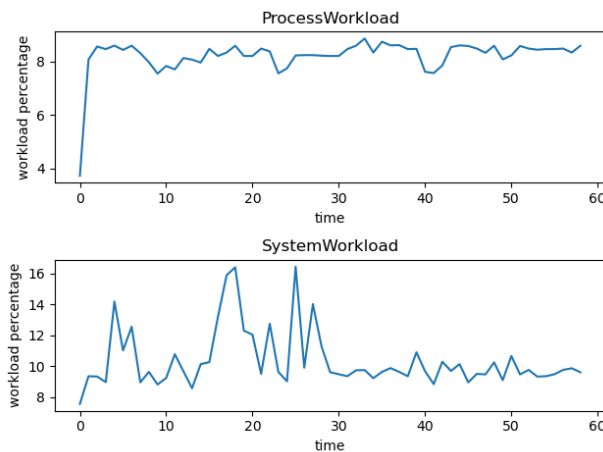Figure 4.11: Task manager priority level

Figure 4.12: Overtime statistics: critical mode

4.2 Findings

4.2.1 Linux

Both critical and normal mode were successfully comparable to *top_command* pre-installed on every Unix machine. One can see that in normal more the process has a priority of 20 and a nice value of zero, which is standard on my Linux machine, while in critical mode the priority is set to RT (which stands for real-time) and the nice value is -19 (the highest nice value on a Unix machine).

While testing the normal mode, one could observe the CPU's percentage exceeding the 100% mark and reaching up to 360%. This is due to the command adding each CPU's workload. Hence the result will show the number of CPUs that the machine has multiplied by the load that each CPU has. In our example we set the system's workload to 90%. That means (in theory) each CPU will have to work 90 percent of the time, which results the load on the whole system would be:

$$number\_of\_CPUs(4) * workload(90\%) = top\_command\_display(360\%)$$

This can also be seen in the critical mode test, where the system's workload was displayed as being almost 40% when we set our workload to 10%.

Additionally the expected system's workload remained in a reasonable range of $10 \pm 1.5\%$, while the process had a range of $10 \pm 1\%$ in the critical mode, while in normal mode the results are similar. Because of the low workload the process's NIT dropped a few milliseconds, which needs to be expected when the thread work only for 10% of the time and for the rest 90% they sleep.

4.2.2 Windows

On my Windows machine the results were resembling those on my Unix machine. The library's output was almost identical to that created by the *task manager*. Similar to Linux the NIT on Windows also dropped a few milliseconds in critical mode. However the difference is substantial between the two operating systems. This might occur, because on Linux we also change the way the process is being scheduled, while on Windows we always use the same scheduler and increase only the priority of the test process. Besides that, the results can be considered satisfying. The workload's range and the process's priority have been pleasing.

5. Future Development

Looking back at the beginning, the project started with small goals, which during the research and development phase, proved to be more complex and have a much higher scalability. Despite this fact the first steps were successfully taken and this showed that the project is not only a dream, but can become reality. With all this said, there is still room for improvement. In time, older tools will change or disappear and newer ones will take their place. Because of that the maintenance will be crucial if this project will live on for more time. At the time of writing I can think only of the following points that will need some improvements:

1. CMake compiler support: although CMake recognizes the compiler by itself, there are a lot of cases where teams develop their software for more than one compiler on the same machine. Therefore it would be a convenience to have a CMake file template that searches for compilers on the calling machine and sets the appropriate CMake variables

2. Windows Fibers support: in my research about threads for different architectures, windows fibers were mentioned. According to Microsoft Docs "fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads"[35]

3. Error Handling: as of now, personally, I'm not satisfied by the way some functions handle errors. This part of the library should also be looked upon by more than one developer, ideally experienced in software development and be improved considering the others feedback

4. Naming conventions: some methods have similar names, because they do almost the same thing, although in some cases a more creative name would be more pleasing

5. CPU's affinity: this subject should be researched more and be also implemented for other operating systems. As of this writing, only the windows part of this library implements a function that manipulates the CPU's affinity for a give process

6. Linking: the current state of the library links its components statically. If possible, the components should be linked dynamically, in order to minimize the size of the calling executable

Bibliography

[1] Edward Snowden. Permanent record. pages 39–40. Pan Mcmillan, 2019.

[2] Dongdong Lu, Jie Wu, Yongxiang Sheng, Peng Liu, and Mengmeng Yang. Analysis of the popularity of programming languages in open source software communities. In *2020 International Conference on Big Data and Social Sciences (ICBDSS)*, pages 111–114, 2020.

[3] D. Spinellis. Choosing a programming language. *IEEE Software*, 23(4):62–63, 2006.

[4] Davide Arcelli, Vittorio Cortellessa, and Daniele Di Pompeo. Performance-driven software architecture refactoring. In *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 2–3, 2018.

[5] Jasmine K.S and R. Vasantha. Identification of software performance bottleneck components in reuse based software products with the application of acquaintance-ship graphs. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pages 34–34, 2007.

[6] Kamil Jezek and Jan Ambroz. Detecting incompatibilities concealed in duplicated software libraries. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 233–240, 2015.

[7] Karuturi Sneha and Gowda M Malle. Research on software testing techniques and software automation testing tools. In *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, pages 77–81, 2017.

[8] Shubhangi Yadav Preeti Wadhwani. Software Market Insights software testing market. https://www.gminsights.com/industry-analysis/software-testing-market, September 2021. Accessed: 2021-11-06.

[9] George Candea, Stefan Bucur, and Cristian Zamfir. Automated software testing as a service. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 155–160, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Kevin J. Valle-Gómez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and José Magallanes-Fernández. Software testing: Cost reduction in industry 4.0. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, pages 69–70, 2019.

[11] Edward F. Miller. Some statistics from the software testing service. *SIGSOFT Softw. Eng. Notes*, 4(1):8–11, January 1979.

[12] Vasundhara Bhatia, Abhishek Singhal, Abhay Bansal, and Neha Prabhakar. A review of software testing approaches in object-oriented and aspect-oriented systems. In M. N. Hoda, Naresh Chauhan, S. M. K. Quadri, and Praveen Ranjan Srivastava, editors, *Software Engineering*, pages 487–496, Singapore, 2019. Springer Singapore.

[13] SPEC Organization. Standard performance evaluation corporation. https://www.spec.org/spec/glossary/#benchmark. Accessed: 2021-11-29.

[14] Klaus Luther. Embedded tutorial: Ic test cost benchmarking. In *12th IEEE European Test Symposium (ETS'07)*, pages 200–200, 2007.

[15] Maya Daneva. *Software benchmark design and use*, pages 20–29. Springer US, Boston, MA, 1995.

[16] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Workloads*, pages 185–201. Springer International Publishing, Cham, 2020.

[17] Vishal Anand and Deepanker Saxena. Comparative study of modern web browsers based on their performance and evolution. In *2013 IEEE International Conference on Computational Intelligence and Computing Research*, pages 1–5, 2013.

[18] Cpp Reference list of c++ standard library headers. https://en.cppreference.com/w/cpp/header. Accessed: 2021-11-06.

[19] Createprocess function. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa. Accessed: 2021-11-12.

[20] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, 2010.

[21] Process identifier. https://en.wikipedia.org/wiki/Process_identifier. Accessed: 2021-11-12.

[22] Kaiwan N Billimoria. *Hands-On System Programming with Linux*. Packt Publishing, 2018.

[23] Anthony Williams. *C++ Concurrency in Action, Second Edition*. Manning Publications, 2019.

[24] Raghu Bharadwaj. *Mastering Linux Kernel Developement*. Packt Publishing, 2017.

[25] Getpriorityclass function. https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getpriorityclass. Accessed: 2021-11-15.

[26] Jospeh Yiu. The definitive guide to arm cortex-m3 and cortex-m4 processors, 3rd edition. Newnes, October 2013.

[27] Shunguang Wu Vardan Grigoryan. *Expert C++*. Packt Publishing, 2020.

[28] std::atomic. https://en.cppreference.com/w/cpp/atomic/atomic. Accessed: 2021-11-17.

[29] Fedor Pikus. C++ atomics, from basic to advanced. what they really do? Accessed: 2021-11-17.

[30] GNU Organisation. Gcc. https://gcc.gnu.org/. Accessed: 2021-12-11.

[31] MinGW Organisation. Mingw-w64. https://www.mingw-w64.org/. Accessed: 2021-12-11.

[32] Kitware. About cmake. https://cmake.org/overview/. Accessed: 2021-12-11.

[33] Michael Kerrisk. sched_setattr(2)-linux manual page. https://man7.org/linux/man-pages/man2/sched_setattr.2.html. Accessed: 2021-11-25.

[34] Michael Kerrisk. proc(5)-linux manual page. https://man7.org/linux/man-pages/man5/proc.5.html#NAME. Accessed: 2021-11-25.

[35] David Coulter Drew Batchelor Michael Satran Karl-Bridge-Microsoft, Vent Sharkey. Fibers. https://docs.microsoft.com/en-us/windows/win32/procthread/fibers. Accessed: 2022-01-06.

List of Figures