# Bachelorarbeit

Hochschule
**Augsburg** University of
Applied Sciences

**Fakultät für Informatik**

Studienrichtung

Technische Informatik

Florian Pîrvu

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

Erstprüfer: Prof. Dr. Thomas Kirchmeier
Zweitprüfer: Prof. Dr. Hubert Högl

Abgabedatum: 20.01.2022

Hochschule für angewandte
Wissenschaften Augsburg

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info(at)hs-augsburg-de

Fakultät für Informatik
Telefon +49 821 55 86-3450
Fax     +49 821 55 86-3499

Verfasser der Bachelorarbeit
Florian Pîrvu
Asternweg 2
86399 Bobingen
Telefon +49 176 3693 7974
pflorian306@gmail.com

# Contents

# 1. Introduction

## 1.1 Motivation

### 1.1.1 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[1]. To make sure fewer problems occur, one could adopt methods like the *Software Performance Engineering*, which uses quantitative techniques to identify designs with lesser flaws and so spare the developers significant time in their implementation[2]. 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 an approach 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 commands 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"[3]. A successful test can be achieved when its main requirements are met. Some examples would be the execution on different environments, time constraints or delivering expected outputs for randomly chosen inputs. Conditions are set by the tester and may vary. These are frequently chosen based on user cases and logical expectations.

### 1.1.2 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[4]. 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, 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 it has grown so big that it would be foolish to ignore it. 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.[5]
*"People may lie, but number don't"*. 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.[6]

### 1.2.1 The costs of perfection

People 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[7].
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. The bad part though, is the pricing, which may or may not be a problem for a company (depending on its size) and additional privacy issues, because we

**U.S. Software Testing Market Size, By Functional Testing, 2020 & 2027 (USD Million)**

755.1

2020                    2027

- System testing   ■ Unit testing   ■ Integration testing
- Smoke Testing   ■ Regression Testing   ■ Others
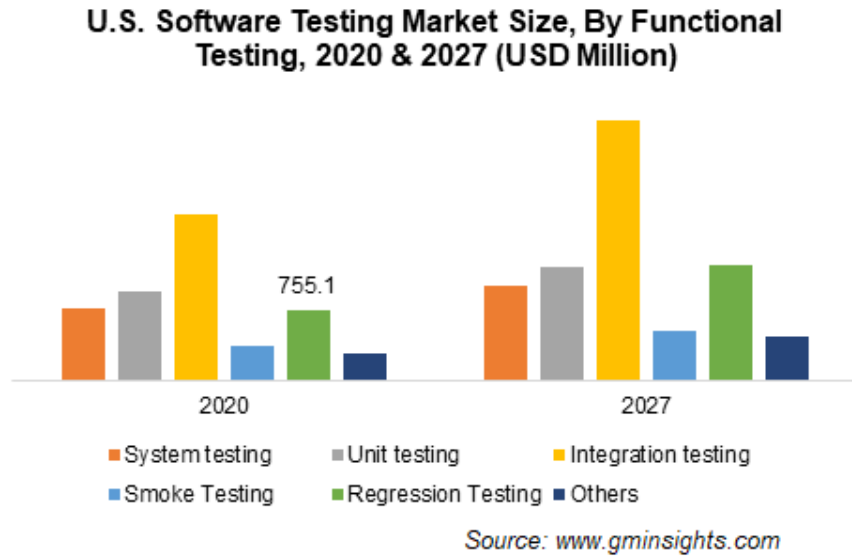
Source: www.gminsights.com

Figure 1.1: US Software Market Size[5]

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.

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 and 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 it for the current tasks,but also for the future ones. Unfortunately when people test internally, additional costs appear. At the begging it may be more expensive than buying a testing software, but in the long run will be cheaper as the testing process gets more and more refined. 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.

It was proven that the cost of finding an error is about \$50 on average [8], 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.[9] 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 [9]:

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 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"[10]. A method developed by *Infineon Technologies AG* ,written by Lukas Klaus, explains "The intention [...] [to] keep cost of test constant relative to overall cost of goods sold". Klaus mention that once the yearly costs of production were established, once 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 in an indicator, which can be calculated by dividing test costs by the revenue of sold goods[11]. This can be used to help decisions on which part is worth improving in order to create the best product while maximizing customer satisfaction.

This leads to 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[12]:

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 1.2. 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)

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.
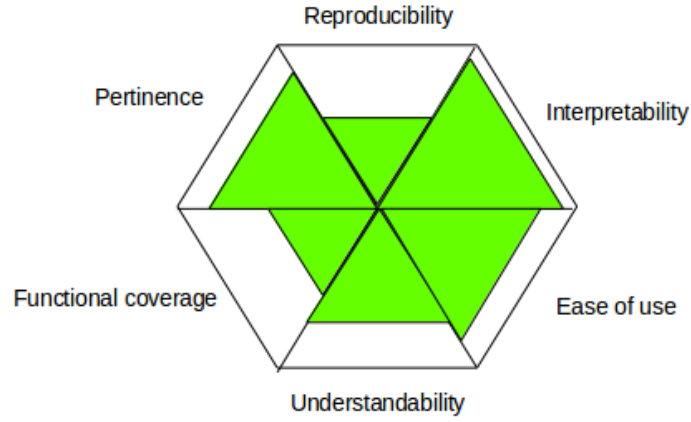
Figure 1.2: Hexagon of criteria

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, unde different configurations and workloads" while natural ones often tend to need user input, cannot be entirely reproduced and often 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[13]. 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 $t_{wait} = 2$s and then an interactive visual window will appear, which will wait for the user to enter some equation. After the equation was typed at $t_{input} = 5$s and the "Enter"-key was pressed, the application proceeds calculating and delivers the answer at $t_{done} = 6$s. 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 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 3.2.2.

## 1.4   Obective

The purpose of this thesis is the development, implementation and testing of a Library for C++ applications that creates an artificial workload and delivers system independent statistics based on multi-threading. This will allow developers to implement their own stress tests and run simulations with different workloads, process priorities and scheduling, which can be used in different manners to achieve real-time applications and create benchmarks for each new product.

# 2. State of the art

## 2.1 Current Situation

At the time of writing this (December 2021), there are four 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 systems 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, but 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, which 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 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 [15] 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?.

---

[0]Some companies also use ARM(aarch64), but this is not an OS and per default doesn't come with any process control system. In order to do so one needs to write or use an extra library for that solely purpose
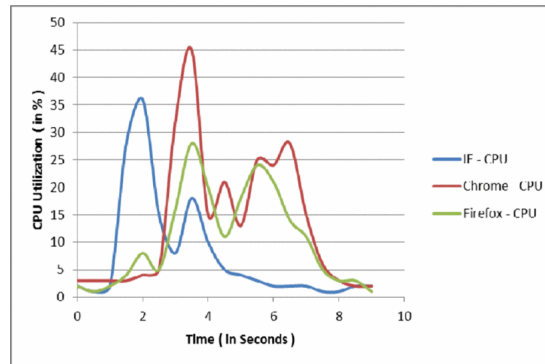
Figure 2.1: Browser CPU utilization comparison[14]

## 2.2   Processes

To many operating systems a process is like an 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 [16], which returns a HANDLE (the equivalent PID for windows systems) and on linux you can use `fork()`[17]. 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.[18][2].

### 2.2.1   Threads

Each process has at least one thread of execution called the main thread, which as the name say contains the `main()` function. Once a thread has been created, the main thread has to wait for it to finish by calling a method called `join()`. As their creators, threads also have unique identifiers called *Thread identifiers* or TID and OS-specific methods to create them. On UNIX systems one would use the `pthread_create()` and on windows `Create_Thread()`. But the role of this library is to make this whole process as easy as possible, so we will use the C++ Standard Library's threads (std::thread) and "detach"[3] ourselves from the other ones.

Processes have their own stack, so they can't communicate with each other. This is a huge problem when it comes concurrency (explained in **??**), 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.

---

[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)

[3]There is also a method called `detach()` which allows a thread to separates itself from the main thread. This way the thread can terminate itself and the main thread doesn't have to wait for that thread's termination

Another difference is that a process can have multiple threads attached to it, but a thread cannot belong to more than one process.[4]. 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.

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 more than one octopus.

In this library we use multiple threads to simulate a user specific workload 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.

### 2.2.4 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"[19].

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



Figure 2.2: Stack segment
[19]

tions there. But 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 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()`.
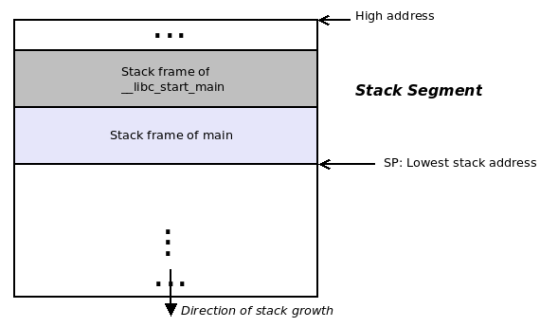
---

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

## 2.3   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[20, 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".

### 2.3.0.1   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 happens 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[21]. The reordering of tasks after switching CPUs will be explained in detail in 2.5.

## 2.4   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).

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.

---

[5]This is a part of context switching, which was discussed in chapter 2.3

## 2.5 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 the its thread priority.

### 2.5.1 Windows

Based on the winAPI documentation[22], 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[6] 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 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.

### 2.5.2 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.

---

[6]If a process is preempted that means it stops executing and yields the cpu

### 2.5.2.1  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*. The RLIMIT_NICE is resource on your UNIX machine and can be set/gotten with the methods `getrlimit()` and `setrlimit()` respectively. These functions takes 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 << "RLMIT_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
RLMIT_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" or log in as root,

use the library's functions, build the program and set the SUID as root. [7]
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 programs.

### 2.5.2.2   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

5. SCHED_RR

The difference between SCHED_RR and "Round Robin Timeshare"(SCHED_OTHER) is
that the realtime policy lets us to coordinate the priorities for that schedueling policy's
queue.
The 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 pro-
cess <=> lowest value).  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.
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>`.[8]
On my Linux Notebook I have the following values: 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 process.
As you can observer 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, yield_sced() 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 of the job expires.

## 2.6   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

---

[7]The SUID is a special bit that one can set and allows normal users to run the program as they were
root
[8]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 2.4: sched_prio_range

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()`. 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: Here
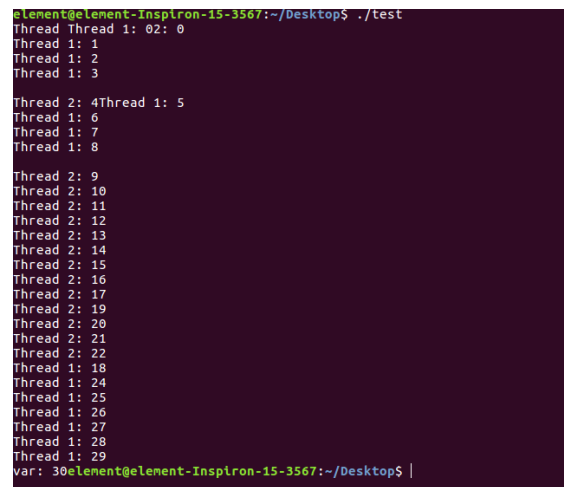
```cpp
1  #include <iostream>
2  #include <thread>
3
4  void addition(int& var, int thread)
5  {
6          for(int i=0; i<10;i++)
7          {
8                  std::cout << "Thread " << thread << ": " << var << std::endl;
9                  var++;
10         }
11 }
12
13 int main()
14 {
15         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 2.5: Thread's behavior without synchronization - Source code

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, these 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 2.6: Thread's behavior without synchronization - Output

## 2.6.1 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()`[20]. 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*)[23].

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*.

## 2.6.2 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[24, 25].

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.

# 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.

## 3.1 Workload

This part of the library is mainly a C++ class, that lets the user create a specific workload.

```cpp
19  namespace workload
20  {
21  class Workload
22  {
23    private:
24        std::vector<std::thread> m_threadList;
25        std::atomic_bool m_runningSimulation, m_startSimulation;
26        std::atomic<uint32_t> m_workload;
27        // Device dependable
28        unsigned int          m_deviceThreads = std::thread::hardware_concurrency();
29        bool                  m_asyncWorkload;
30
31    public:
32        Workload(unsigned int workloadPercentage, unsigned int* WLPriority, bool async);
33        ~Workload() = default;
34        void finishWorkload();
35        void simulateWorkload(unsigned int randomize, unsigned int* WLPriority);
36        void setWorkload(uint32_t workload);
37
38        // Simulation switch
39        void startWL();
40        void stopWL();
41  };
42  }
```

Figure 3.1: Thread's behavior without synchronization - Source code

### 3.1.1 Attributes

The class has six private Attributes:

1. threadList: is a list of type vector that holds the current number of working threads after the 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 `simulate-Workload()`). This is in order to synchronize threads and start them simultaneously

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 in \*section\*.

The name start with "m_" in order to tell the programmer that the variable is a member of the class. It makes it easier to read and informs the user about the variable 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: given workload percentage, that the user wants 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: let s the user to create a more "real" workload. This will be explained in detail in chapter\*\*\*

   The first thing the constructor checks when a new object is created is the validity of the passed workload. Next it sets the *runningSimulation* and *startSimulation* attributes to false for a synchronized start. Afterwards it measures the current workload for ten seconds (it can be 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 the loop creates the threads with the class's `simulateWorkload` method.
   `emplace_back()` pushes the created threads into an `std::vector`. I opted for this method instead of `push_back()` because the compiler doesn't have to create the thread 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

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: is the index in the thread's construction loop

   (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 `startSimu-lation` 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. This way each thread starts with a small delay so the library's workload will never be zero. One can imagine a sinus curve if the flag is set and a square wave function otherwise (as seen in Figure 3.2). Afterwards the actual workload loop will start. I decided to recalculate the sleep
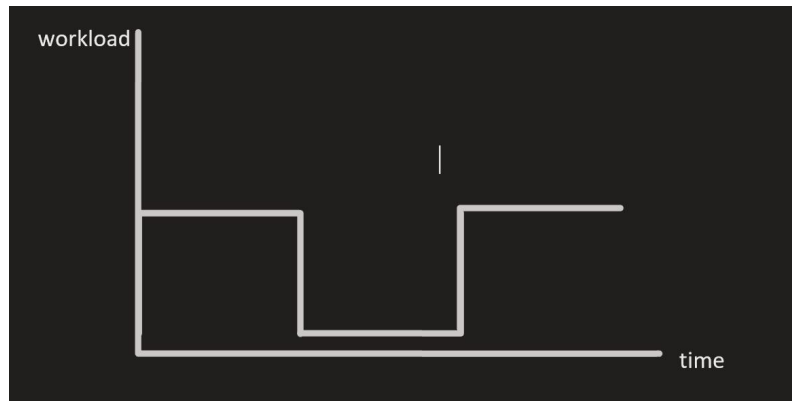


(a) `asyncWorkload` not set
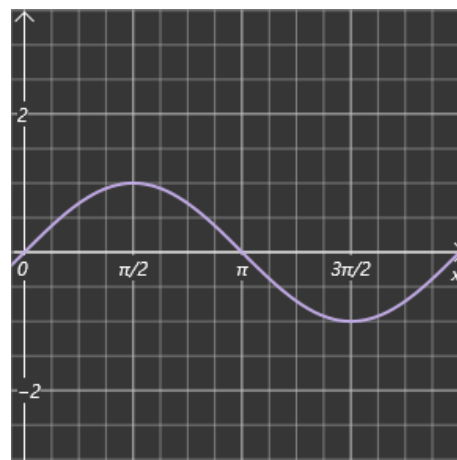


(b) `asyncWorkload` set

Figure 3.2: Workload Time Diagrams

time based on the current workload here, in order to support future dynamically changing workload for the simulation. The frequency of the 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 a process can run on (see the cpu's affinity 2.4 and the scheduler's policy 2.5.2.2).

### 3.2.1 Definitions

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 2.5.1)

6. threadPrioList: vector containing thread priorities for a WINDOWS machine in ascending order(see 2.5.1)

7. sched_attr: struct containing thread's attributes for a LINUX machine

### 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 and 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.

(d) decreaseSchedClass(): does almost the same as `increaseSchedClass()`, but instead of increasing the scheduler class's priority, it decreases it

2. Linux/Unix:

(a) readCPUTime(): this functions takes an argument of type integer and reads data about the CPU specified by the given number. 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[26]:

| name | user | nice | system | idle | iowait | irq | softirq | steal | guest | guest_niced |
|------|------|------|--------|------|--------|-----|---------|-------|-------|-------------|

The most important statistics for the library are[26]:

   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 `readData()` 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 2.5.2.2.

(d) changedPolicy(): 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.[27]

(e) decreaseProcessNiceValue(): this method takes an integer as its argument specifying the 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/increments 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 former mentioned function, but instead of decreasing/incrementing the value this method increases/decrements it.

3. NON-OS-Specific:

   (a) getCPUTime(): 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).

4. 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 when this might be important).

On Linux the function calls the `times()` method specified by 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).

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

5. 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 is 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.

---

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

# 4. Beispiele

Bla fasel. . .

Beispiele

## 4.1 Zitieren

Quellen[?, ?, ?, ?, ?] nicht vergessen. Dazu verwendet ihr bibtex.
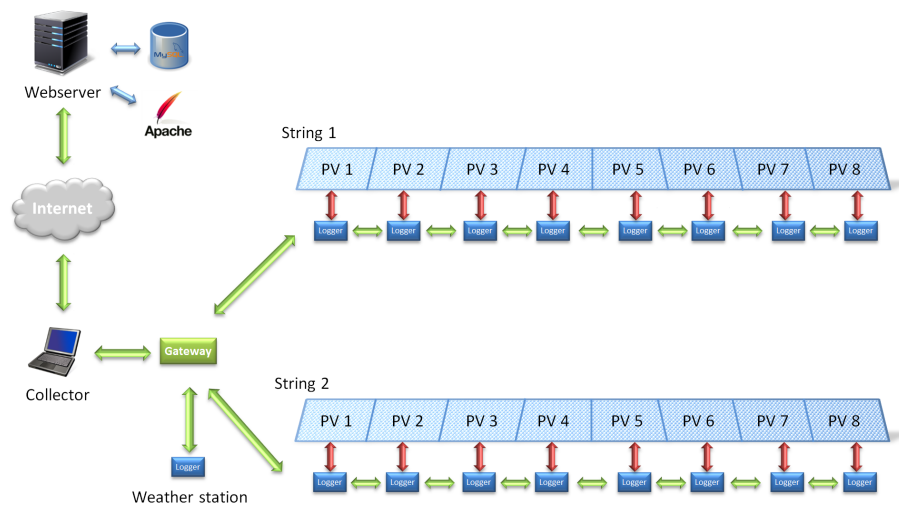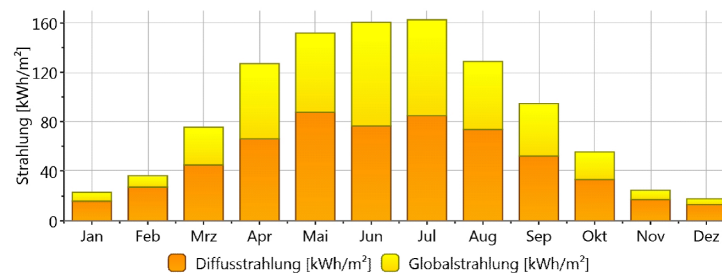
## 4.2 Bild einfügen

### 4.2.1 Ein Bild skaliert

Figure 4.1: Beschriftungstext

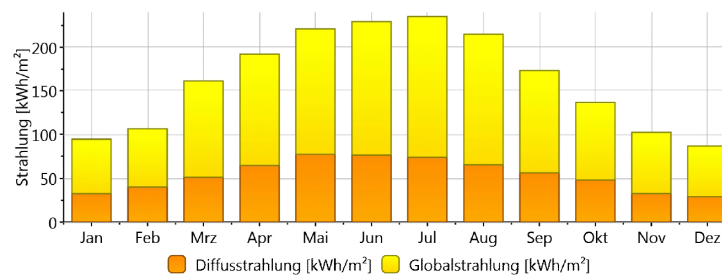### 4.2.2 Zwei Bilder nebeneinander oder untereinander

## 4.3 Tabellen

Globalstrahlung monatlich

(a) Beschriftung Bild links

Globalstrahlung monatlich

(b) Beschriftung Bild rechts

Figure 4.2: Beschriftung beide Bilder

| Firma | Produkte / Lösungen | WEB |
|---|---|---|
| Concentrix (Soitec) | Module mit Konzentratoren (Fresnel-Linsen) | http://www.soitec.com |
| Isofoton | Module mit Konzentratoren (Fresnel-Linsen) | http://www.isofoton.com |
| Semprius | Module mit Konzentratoren (Fresnel-Linsen) | http://www.semprius.com |
| Azur Space | Mehrfach Junction Zellenhersteller | http://www.azurspace.com |
| Cyrium Technologies | Mehrfach Junction Zellenhersteller | http://www.cyriumtechnologies.com |
| Emcore | Mehrfach Junction Zellenhersteller | http://www.emcore.com |

Table 4.1: Hersteller von CPV-Produkten

Table 4.2: Single-hop Scenario - Traffic Pattern

| Pattern | Parameter | Distribution | Range/Values |
|---|---|---|---|
| **Burst** | Burst IAT | uniform | [9.9; 10.1] s |
| | Packets per Burst | constant | 100 |
| | Packet IAT | constant | 0.02 s |
| | Packet Size | constant | 1024 bit |
| | # Sources | - | 2 |
| | Offset | uniform | [0; 1] s |
| **Single** | Packet IAT | uniform | [0.9; 1.1] s |
| | Packet Size | constant | 1024 bit |
| | # Sources | - | [10;20;30;40;50; 60;70;80;90;100] |
| | Offset | uniform | [0; 1] s |

# 5. Analyse

Bla fasel...

## 5.1 Abschnitt 1

Bla fasel...

## 5.2 Abschnitt 2

Bla fasel...

### 5.2.1 Unterabschnitt

Bla fasel...

#### 5.2.1.1 Unter-Unterabschnitt

# Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

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

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

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

[14] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

# List of Figures