02159 - Operating Systems
OS Challenge

Document version 2023-02

1 Introduction

This is the **mandatory** group project for the course 02159 - Operating Systems. The project has the form of a challenge or a friendly game. This document specifies the objective and rules of the challenge, as well as details about the assessment and its contribution towards the final grade. Note that **this document is preliminary and may change**. Pay attention to the document version and always make sure that you have the latest version. Any updates will be announced on DTU Learn.

1.1 Important Dates

Here, you can find a summary of the deadlines and milestones for quick reference.

- **Challenge Kick-off: August 31, 2023**
- **Group Information Submission: September 11, 2023**
- **Milestone Submission: October 2, 2023**
- **Final Submission: November 15, 2023**
- **Group Presentations: November 30, 2023**
- **Announcement of Winners: November 30, 2023**

1.2 Learning Objectives

Operating Systems (OS) are performance-oriented systems. They are responsible for managing the resources of a computer system (processing power, memory, energy, etc). These resources are not infinite, and it is the role of the OS to manage them efficiently and effectively. A solution that only works is not enough; instead, we need solutions that work fast.

In this spirit, the intention of this project is to go beyond student assignments that must simply work. You will face a challenge that is (relatively) easy to solve in a quick and dirty way. Yet, it leaves you a lot of room for performance improvements. Your end goal is not to create something that works, but **something that works fast**. The fastest team will win the challenge.

You will create a server application in the **C programming language**. The server will operate at the interface with the **Linux OS**, therefore you will learn to use Linux system calls, and in general, to program at the OS interface. The challenge itself is a dummy application that emulates challenges that an OS has to deal with. To provide a fast solution you will have to learn how the Linux OS works and how you can use it get the maximum performance. Moreover, you will have to understand how an OS addresses similar challenges. This will give you ideas on how to solve your challenge and make your server faster.

There are often multiple ways to solve the same problem. Reading books, papers, or articles on the Internet are good for getting ideas, but is rarely sufficient for determining which solution is the fastest. For that you will need to get evidence through **experiments**. With this project, you will also learn how to conduct experiments to compare solutions and determine which solution should be part of your final submission.

1.3 Document Structure

- **Section 1** summarises the deadlines and learning objectives of the challenge.
- **Section 2** provides the specification of the server that you need to develop, the provided tools, and the general rules for the challenge.
- **Section 3** provides details on the logistics and the overall process of the challenge, such as group formation, deliverables, ranking, presentation and assessment.
- **Section 4** provides details for developing and executing your server.
- **Section 5** summarises what you need to submit for each deadline, so that you can easily make sure that you did not forget anything.
- **Section 6** answers a number of frequently asked questions.
- **Section 7** summarises this document's version history. If a new version is released, a summary of changes will be available in this section.

2 Challenge Specification

2.1 Server Overview

You will create a server that accepts requests for **reverse hashing**. A cryptographic hash algorithm is a one-way function. It takes an arbitrary input message and converts it to a byte sequence of a fixed size: the hash or message digest. A hash function has two important properties. Firstly, it is **deterministic**. This means that it always produces the same output for the same input. Secondly, it is a function that it is practically infeasible to invert in a fast way. In other words, it is computationally easy to get the hash from the input, but computationally very hard to get the original input from the hash (reverse hashing). Unless the cryptographic hash algorithm is compromised, the only way to find the original message that produces a given hash is through a **brute-force search**: try out all the possible inputs and check if they produce a match.

Your server will accept requests messages that include a hash generated by the **SHA256 algorithm**. SHA256 generates hashes of 256 bits (32 bytes). The input message that produces the hash is a 64-bit unsigned integer. This is a number from `0x0` to `0xFFFFFFFFFFFFFFFF` in hex format. Brute-forcing 2^{64} numbers would take a really long time. Therefore, the requests also include a `start` and `end`. These two numbers specify bounds that make the search more feasible: the input number that you are looking for is in the range `[start, end)`. Your goal is to find the number that generated the given hash by generating the hash of all the numbers from `start` to `end` until you find a match, and report it back to the client with a response message. You are expected to use the OpenSSL library that incorporates a SHA256 implementation.

Your server must be able to handle **multiple requests** and respond to all of them as fast as possible with the correct answer.

2.2 Protocol and Packet Specification

Your server will accept requests from a request generator (client) that will be provided to you. The server must not assume a specific IP address and, instead, it should bind to any IP address (`INADDR_ANY`). The communication between the client and the server must adhere to the rules specified in this section. This protocol builds on top of **TCP sockets** and operates as follows. Each reverse hashing request is communicated over a TCP connection. Upon the establishment of a TCP connection the client sends a **request packet** to the server that is exactly 49 bytes. The server responds to the client with a **response packet** that is exactly 8 bytes. The TCP connection is then terminated and the TCP socket is closed. Each TCP connection handles a single reverse hashing request. Multiple TCP connection requests may arrive at the same time.

The request packet has 4 fields as specified in Figure 1. The first 32 bytes correspond to the `hash` that needs to be reversed (the type is an `uint8_t` array). The following 8 bytes correspond to the `start` (the type is `uint64_t`). The following 8 bytes correspond to the `end` (the type is `uint64_t`). Finally, the last byte, `p`, correspond to the priority level of the requests (the type is `uint8_t`).

The response packet has 1 field as specified in Figure 2. The 8-byte field corresponds to the `answer`, that is the number that generated the `hash` in the request (the type is `uint64_t`).

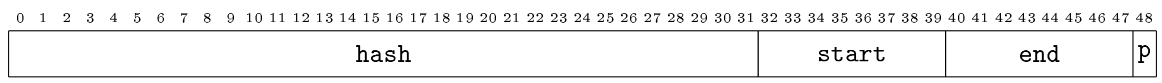

Figure 1: Format of a request packet.

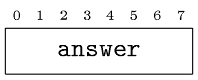

Figure 2: Format of a response packet.

You have to pay careful attention to the **byte order**. In particular, the byte order of all packet fields in transit (i.e. `hash`, `start`, `end`, and `answer`) are **big-endian** (also known as network byte order). However, the byte order of the 64-bit unsigned integer that is used as input to generate the `hash` at the client is **little-endian**. You may use the functions `htobe64`, `htole64`, `be64toh`, and `le64toh` to convert between byte orders and ensure that you always use the correct one. You are provided with the header file `messages.h`[1] that provides definitions that correspond to the formats of the two packets, as well as definitions of the above functions for MacOS.

2.3 Server Input Arguments

The server executable must be named `server` and it should accept exactly one input argument, the TCP port number that the server is listening at. Your server will be executed by a script, as shown below.

```
$ ./server port
```

2.4 Request Generator (Client)

You are provided with a request generator[2] (client) that you can use to test your server. The client generates reverse hashing request in accordance with the input specified in the command line arguments. Specifically, the generator requires 9 command line arguments, as shown below.

```
$ ./client hostname port seed total start difficulty rep delay lambda
```

2.4.1 Input Parameters

- `hostname`: Host name or IP address of the server. You may use `localhost` if both client and server run at the same computer.
- `port`: Port number of the server.
- `seed`: Seed for the random number generator. If set to 0 the current time will be used as seed.
- `total`: Total number of reverse hashing requests to be generated. Must be positive integer.
- `start`: Hashes will be generated from input numbers that are greater than or equal to `start`. If set to 0, the value of `start` will be randomised on each request. Must be positive integer.
- `difficulty`: Hashes will be generated from input numbers that are less than `start`+`difficulty`. Must be integer and greater than zero.
- `rep`: Repetition probability percentage (%). Must be between 0 and 100. If set to 0, hashes will be generated from random numbers following a uniform distribution (all values in the range `[start,end)` have equal probability to occur). If positive, then there is a `rep` percent probability that a previously selected number is repeated. If this happens, all previously selected numbers have equal probability to be re-selected. Otherwise, a uniformly random number in the specified range is selected.
- `delay`: Delay between requests in microseconds (`usec`).

[1]`https://github.com/dtu-ese/os-challenge-common.git`
[2]`https://github.com/dtu-ese/os-challenge-common.git`

- `lambda`: The priority level of each request (see field `p` in Figure 1) is generated randomly, following an exponential distribution with rate `lambda`[3]. There are 16 priority levels from 1 to 16. The higher the number, the higher the priority. If `lambda` is set to 0, all requests will have the same priority level (`p=1`), therefore prioritisation is disabled.

2.4.2 Benchmarks

First and foremost, the request generator confirms that the `answer` is correct. It is also responsible for measuring the time between sending a request and receiving the response from the server (latency, l). These measurements are used to calculate the final score and ranking.

More specifically, the first benchmark is **reliability** and it is measured as the sum of the correct answers over the `total` number of requests. The second benchmark is the **average latency** of the server, calculated as the sum of the latency measurements over the `total` number of requests. In the above summation the delay measurements receive a penalty that is proportional to the priority level (`p`), as specified in the following equation.

$$\texttt{score} = \frac{1}{\texttt{total}} \sum_{i=1}^{\texttt{total}} l_i \texttt{p}_i \quad (1)$$

2.4.3 Output

The request generator initially prints its **version number** and the configuration parameters. In turn, for each response packet that it receives, it prints out a message of the following format.

```
"[%d %llu %llu %u] %llu\n"
```

The first variable is 1 if the `answer` is correct (i.e. generates the `hash`) and 0 otherwise. The second variable is the number that generated the `hash` at the client. The third variable is the `answer` received from the server. The fourth variable is the (randomly generated) priority level, `p`. The fifth variable is the time between sending the request and receiving the response from the server in microseconds (`usec`).

Upon the completion of all requests, the generator prints out statistics and results. If priorities are enabled (`lambda` > 0), it prints out a list of 16 numbers. Each number corresponds to a priority level and in particular to the total number of requests that have the respective priority level. Finally, the two benchmarks, namely the reliability percentage and the `score`, are printed.

2.4.4 Tips and Tricks

The request generator is a tool for experimentation and testing. Here are some ideas on how you can use it.

- You can limit `start` and `difficulty` to eliminate randomness in the requests. For example set them to 1 and 1 respectively and the generator will always generate the same request (*i.e.* the hash value of 1). This is a great way to debug your server. You can also use it to confirm that you correctly read the request packet and write the response packet. You can use it to confirm that you correctly execute the SHA256 algorithm, as you already know the answer. If you have troubles generating the same hash as the client for a known input, it is most likely that the problem is in the byte order (see Section 2.2). Check also the FAQ in Section 6.12.
- The parameter `difficulty` controls the difficulty of the requests. The larger the value, the longer it will take to brute-force the answer. Set it to an easy configuration during development for quick feedback, but remember to test difficult settings as well.
- The `delay` parameter controls the request arrival rate at your server. Try different values in combination with different `total` values. Can you survive big bursts of requests? If your solution works fine with higher values of the `delay` parameter but fails with lower values, it is likely that your server cannot handle many requests in parallel. Check also FAQ in Section 6.13.
- The repetition probability `rep` controls the profile of the requests. Try out the extreme values, 0 and 100, and some values in between. Are the requests totally random or the have some temporal locality?

[3]`https://en.wikipedia.org/wiki/Exponential_distribution`

- The priority levels are randomly selected from an exponential distribution. Depending on the `lambda` parameter, the exponential distribution has higher probability to give smaller numbers and a low probability to give high numbers. Hence, the majority of the requests will have a low priority, but few high priority requests will arrive from time to time. Try out different values, such as 0.5, 1.0, and 1.5 to observe how it behaves.
- You can use a fixed `seed` to eliminate the randomness in the requests. This is a good way to compare solutions and debug. In the final execution the seed will be unknown, so make sure that you try out different seeds to ensure that your solution works well on average despite the specific seed.

2.5 Insights: Alignment with Operating Systems

The challenge is carefully designed to emulate the real-world challenges of a modern OS. The reverse SHA256 plays the role of an intensive task. Similarly to your server, a modern OS must handle tasks from the users. These tasks may arrive sporadically or multiple at the same time in bursts. Some of these tasks are characterised by a temporal locality. For example, a user is more likely to open a recent file than a file that was modified last year. This quality is emulated by the repetition probability, which controls the probability that the same request will be generated again. Lastly, not all requests are of equal urgency. For example, let us assume that an OS needs 1 second to initiate a large file transfer. This delay will have no negative impact to the user experience: the user expects this to take some time anyway. But if the OS needs 1 second to move the mouser pointer, the user experience rapidly drops. This 1 second of latency has a much higher cost. This property is emulated by the priority levels and the penalties in the `score`.

2.6 Rules

The following rules apply:

- Your source code must be written in the **C programming language**.
- Your source code must use POSIX system calls and compile for **Linux**.
- It is OK to use the `libc` and `pthread` libraries.
- You are generally not allowed to use non-standard libraries except OpenSSL for SHA256. If you are uncertain, ask.
- It is OK to use *up to* optimisation level `-O3` for `gcc`. Optimisation level `-Ofast` is not allowed.
- The server will be executed without root privileges.
- You are not allowed to use a pre-calculated hard-coded look-up table of inputs and hashes.
- You are asked to be creative and think out of the box, but make sure that your solutions are aligned with the spirit of the challenge. If you are uncertain, ask.

3 Logistics

3.1 Process

The challenge is inspired by Problem-Based Learning (PBL). PBL is a teaching method that originates from the medical sciences. The idea is that students form groups and are given a challenging problem to solve. The students are responsible for breaking the problem into smaller problems, identify what skills and knowledge they are missing, specify their own personal learning objectives, and work independently towards a solution.

You are expected to start working on the project from day one (see kick-off in Section 1.1). Do not waste time, waiting until the group formation is finalised. Instead, **start working individually as soon as possible**: study the challenge and start analysing it, identify background knowledge that you may miss and start working on it, etc. When groups are finally formed, share the preparation work that you did with your teammates.

In parallel with the challenge, there will also be **weekly supplementary video lectures**. These lectures will be around OS concepts. They will not directly discuss the challenge, yet they will be aligned to the challenge in the sense that deep understanding of the OS concepts discussed in the lectures will you help a lot in designing a fast solution for the challenge.

You have **three submission deadlines**. First, you need to submit a **group information file** (deadline specified in Section 1.1) that follows the format specified in Section 3.2.

For the **milestone** (deadline specified in Section 1.1) focus on creating a server that works. In other words, focus on creating a server that yields a 100% reliability (ignoring the `score` for now). To that end, you may temporarily disregard the repetition probability (set `rep=0`) and the priority levels (set `p=0`). In the provided repository[4], you can find execution scripts with different client configurations. You shall first debug your server on the basic `run-client.sh` and then test it on the more challenging `run-client-milestone.sh`. You are strongly encouraged to test it on **additional client configurations of your own** to identify obscure bugs.

The **final submission deadline** is also specified in Section 1.1. For the final submission you are expected to conduct a series of **experiments**. In other words, you are asked to design, implement, and compare different solutions, in pursue of the fastest. **Each group member is expected to conduct at least one experiment**. An experiment could be anything that makes sense. For example, one experiment could determine if a multithreaded approach is faster than a multiprocess approach. Through these experiments you will determine the best approaches and integrate them into a final solution. For benchmarking and continuous feedback, the binary of a working reference implementation of the server is also provided[5]. As part of your experimentation, you may compare the performance of your server against the reference implementation under the same conditions.

3.2 Group Formation, Group Name, and Git Repository

You are responsible to self-organise into groups. The recommended group size is **2-4 persons**. Groups of 1 person and 5 persons are allowed but not recommended.

After group formation, each group should meet, and specify ground-rules for the collaboration. A recommended model for collaboration is discussed in Section 6.9. In addition, each group should decide a **group name**. You can be creative, but please, let's avoid offensive names. Your group name must start with a letter or number and must not contain the space character.

You shall use the **GitLab** server of the GBAR (`https://gitlab.gbar.dtu.dk/`) for collaboration and source code dissemination. Therefore, you should also create a GitLab repository for your group. The repository must be **private** and named as follows: `os-challenge-<group-name>`. The repository can be hosted under the account of one of the group members. In turn, you should generate a 6-digit secret random number. You will use this secret number to interpret the rankings that will be published anonymously.

Finally, you should also generate a **group information file**. The file should be named `<group-name>.csv` and it should be formatted as a comma separated values text file. The file should include the following entries in this order. The file will be parsed automatically, so it is important to follow the formatting instructions exactly.

- Group Name
- PIN: a 6-digit secret random number
- Link to the GitLab repository
- Number of group members
- Full name of 1st group member
- DTU email of 1st group member
- GitLab username of 1st group member
- Full name of 2nd group member
- DTU email of 2nd group member
- GitLab username of 2nd group member
- ...

[4]`https://github.com/dtu-ese/os-challenge-common`
[5]`https://github.com/dtu-ese/os-challenge-common`

To make sure that you generate your group file according to specifications, you can use the Python script `submission.py`[6].

Important: If you generate your submission files on a Windows machine, make sure that the files are using the UNIX format for newlines. It is easier and recommended using the `submission.py` that does it for you.

```
$ python3 ./submission.py group
**************************
* DTU 02159 OS Challenge *
**************************

Generating a new group...

Group name: xefa
PIN (must be positive 6-digit integer): 675412
Group Repository URL: https://gitlab.gbar.dtu.dk/xefa/os-challenge-xefa
Number of group members: 2
[Student 0] Full Name (as in student card): Xenofon Fafoutis
[Student 0] DTU Email: xefa@dtu.dk
[Student 0] Git Username: xefa
[Student 1] Full Name (as in student card): John Doe
[Student 1] DTU Email: jd@dtu.dk
[Student 1] Git Username: jd

Your group submission string is:
xefa,675412,https://gitlab.gbar.dtu.dk/xefa/os-challenge-xefa,2,Xenofon Fafoutis,
xefa@dtu.dk,xefa,John Doe,jd@dtu.dk,jd

Submission file generated: xefa.csv

$ cat xefa.csv
xefa,675412,https://gitlab.gbar.dtu.dk/xefa/os-challenge-xefa,2,Xenofon Fafoutis,
xefa@dtu.dk,xefa,John Doe,jd@dtu.dk,jd
```

You must **submit the group information file** by the deadline shown in Section 1.1.

3.3 Deliverables

For the final submission, as well as the milestone submission, you will **submit the GitLab repository**, simply by giving read access to the teacher and providing the **git hash** of the milestone and final submission respectively. Make sure that the date of the commit is before the respective deadline. You shall submit the milestone and final commit hash in a `csv` file, named `<group-name>-milestone.csv` and `<group-name>-final.csv` respectively. You can again use the Python script `submission.py`[7], as in the examples that follow:

```
$ python3 ./submission.py milestone
**************************
* DTU 02159 OS Challenge *
**************************

Generating a milestone submission...

Path to your group csv file (e.g. xefa.csv): xefa.csv
Group Name: xefa
Submission Git Hash (e.g. f54c4e538483ecd8bbf5482a986bc13b12639547):
f54c4e538483ecd8bbf5482a986bc13b12639547
```

[6] `https://github.com/dtu-ese/os-challenge-common`
[7] `https://github.com/dtu-ese/os-challenge-common`

```
Your milestone submission string is:
xefa,f54c4e538483ecd8bbf5482a986bc13b12639547

Submission file generated: xefa-milestone.csv

$ python3 ./submission.py final
**************************
* DTU 02159 OS Challenge *
**************************

Generating a final submission...

Path to your group csv file (e.g. xefa.csv): xefa.csv
Group Name: xefa
Submission Git Hash (e.g. f54c4e538483ecd8bbf5482a986bc13b12639547):
f54c4e538483ecd8bbf5482a986bc13b12639547

Your final submission string is:
xefa,f54c4e538483ecd8bbf5482a986bc13b12639547

Submission file generated: xefa-final.csv
```

Important: If you generate your submission files on a Windows machine, make sure that the files are using the UNIX format for newlines. It is easier and recommended using the `submission.py` that does it for you.

The repository should be **neatly organised** and the file and folder names should be intuitive. In addition to your final solution, you should also include in your repository code that corresponds to the **experiments** (even if the ideas tested are not included in the final version of your server) and **documentation** in a `README.md` file.

Important: To the extent that is possible, make sure that the person who commits the code to the repository is the person who wrote the code. As you know, DTU rules dictate individual grading for group projects. Git keeps track of which lines of code were committed by whom, and it will be used to detect free-riders. In the **final submission**, I expect to see **commits from all team members**, but you don't need to get paranoid. Grades will **not** be proportional to the lines of code. To this end, you should also make sure that your commits have your name and DTU email address. You can edit those by executing the following configuration commands before you commit your code:

```
$ git config user.name "Xenofon Fafoutis"
$ git config user.email "xefa@dtu.dk"
```

3.3.1 Repository Organisation

The submitted (first milestone and then final) solution should be placed at the git root directory and at the default branch. You may organise your source code is subdirectories as you see fit. Make sure that all source code can be simply compiled with a `Makefile`[8] at the git root directory. The executable, must be named `server` and it should also be generated at the git root directory. The `Makefile` should be called `Makefile` with capital `M`. Your server will be cloned, compiled and run as in the following example:

```
$ git clone <your-git-repo>
$ git checkout <milestone/final-git-hash>
$ make clean
$ make
$ ./server <port>
```

The code of experiments should be placed either in separate directories or in separate git branches. Also, make sure that you comment your code.

[8] `https://www.tutorialspoint.com/makefile/index.htm`

Your repository should also contain a `README.md` file. In the `README.md` file, you will briefly present your experiments and the final solution. Also, describe the **folder structure** and **branch structure**, making sure that it is clear where the reader can find what. For the final report convert the `README.md` into a `pdf` file and submit it on DTU Learn.

3.4 Ranking and Presentation

The submitted solutions will be compared under the same conditions in the execution environment (see Section 4.4). The **configuration parameters** are as specified in `run-client-final.sh`, apart from the `seed` which will remain secret. Note that `total` might change depending on the number of groups. In the final ranking, the teams will be first ranked according to reliability and then according to the median `score`. The **winning team** will be the one with the lowest median `score` and 100% reliability.

At the end, **all groups** will present their experiments and final submission in the classroom. **Each group member** is expected to briefly discuss their experiment(s). There will also be an opportunity for questions from the audience. After the presentations, the **results will be announced**. The winning team and the two runner-ups will be announced and congratulated. All other groups will be anonymised and notified about their performance in private.

3.5 Assessment

The final submission and presentation are mandatory and count towards the final grade, which will be determined with overall assessment. **It is highlighted that your final grade will be determined by how well you meet the learning objectives and not by the rankings**. Indeed, the assessment will be based on absolute criteria and not relatively to the performance of others. The friendly competition is just for fun.

4 Development and Execution

4.1 Development Environment

For development, you are free to use any tools and environments you prefer.

Remember, however, that the source code that you deliver needs to compile (without errors or warnings) and run correctly in the execution environment (Section 4.4). Your server **must compile** (without errors or warnings) and **run correctly** in the execution environment.

You are provided with two Vagrant[9] Linux Virtual Machines: one for the client and one for the server. These VMs, running on a DTU Compute Linux server with a x64 processor will be the **execution environment**. You can also host the VMs in your own machine and use them as a **testing environment**.

For development, you ideally need access to a Linux machine with x64 processor. If you don't have access to a Linux machine, you can also use the provided Vagrant Linux Virtual Machines for development, which are compatible with Windows and macOS machines with a x64 processor. The Vagrant VMs are set up with a synchronised folder between the host and virtual machine to facilitate software development.

The Vagrant Virtual Machines of the execution environment are unfortunately not compatible with ARM64 processors, including macOS based on Apple Silicon. If that's your computer, see Section 4.5 for workarounds.

Whatever your development environment, make sure that your code compiles and works correctly in the exectution environment before you submit it.

4.2 Getting Started

The best way to address big problems is by breaking them into smaller problems. Divide and conquer. To that end, your first step is to make a server that works reliably. Once you have something that works, you can then focus on making it fast. Before you start coding, make sure that you refresh your skills in C programming[10], the Makefile[11] building tool and in the UNIX command line interface[12].

[9] `https://www.vagrantup.com/intro/index.html`
[10] `https://www.tutorialspoint.com/cprogramming/index.htm`
[11] `https://www.tutorialspoint.com/makefile/index.htm`
[12] `http://www.ee.surrey.ac.uk/Teaching/Unix/`

You are asked to create a TCP server in C. Your first step should be to understand how TCP sockets work in Linux and how to make a basic client/server application[13]. You can then use the request generator (client) to create predetermined requests by limiting the `start` and `difficulty` parameters. If you limit them to generate requests of always the same input, then you know the `answer` without the need to execute SHA256 and look for a match. If you have troubles with the communication protocol and the packet formats, you can use a network traffic analyser, such as Wireshark[14], to inspect the packets that you receive and transmit.

The next step would be to execute the SHA256 algorithm using the OpenSSL library[15]. Read the manual and understand how the `SHA256_*` API works[16]. OpenSSL is pre-installed in the testing environment. If you use your own system, it is likely that it comes with OpenSSL pre-installed or you can easily install it via a package manager. You can also compile the source code[17] and use the `-I` and `-L` flags of `gcc` to include the header and object files[18].

4.3 Testing Environment

The testing environment is composed of two Vagrant Linux Virtual Machines. Vagrant provides a way to specify recipes for virtual machines. Familiarise yourself with it[19]. The testing environment is compatible with all major operating systems but requires a machine with a x64 processor. If your compute is macOS with Apple Silicon, see Section 4.5 for workarounds.

To set up the testing environment you need to follow the following steps. First, install VirtualBox[20] and Vagrant[21]. Then, clone the `os-challenge-common` repository[22]. Navigate to it. The virtual machines are specified in the `Vagrantfile` in the `x86_64` directory.

```
$ cd x86_64
```

Familiarise yourself with the command line interface of Vagrant[23]. To start them you need to add the Ubuntu Server box:

```
$ vagrant box add ubuntu/focal64
```

In turn, create and configure the guest machines with the following command:

```
$ vagrant up
```

You can now log into the server by typing:

```
$ vagrant ssh server
```

Similarly, you can log into the client by typing:

```
$ vagrant ssh client
```

If it asks for a password, try the default password: `vagrant`. Once inside the server/client machine, notice the directory `os-challenge-common` in the home directory. This is sync'ed with the respective directory of the host machine. It is also added in the `$PATH` variable. Moreover, you should be able to `ping` the server from the client and vice versa.

```
vagrant@client:$ ping 192.168.101.10
```

Tip: Use `lscpu` and `top` to study the CPU and memory resources of the server. Think how you can use this information to optimise your server.

You can then clone your server as follows:

```
vagrant@server:$ git clone https://github.com/<user>/os-challenge-<group-name>.git
```

[13] https://www.tutorialspoint.com/unix_sockets/index.htm
[14] https://www.wireshark.org/
[15] https://www.openssl.org/
[16] https://www.openssl.org/docs/manmaster/man3/SHA256.html
[17] https://github.com/openssl/openssl
[18] https://linux.die.net/man/1/gcc
[19] https://www.vagrantup.com/intro/index.html
[20] https://www.virtualbox.org/wiki/Downloads
[21] https://www.vagrantup.com/downloads.html
[22] https://github.com/dtu-ese/os-challenge-common.git
[23] https://www.vagrantup.com/docs/cli/

You can then start your server as follows:

```
vagrant@server:$ ./server <port>
```

Then, you may start the client on the other guest machine:

```
vagrant@client:$ run-client.sh
```

Tip: Make sure that you start the right server: `./server <port>` will execute the binary `server` in the current working directory, whereas `server <port>` will execute the binary `server` found in the `$PATH`.

4.4 Execution Environment and Continuous Feedback

The execution environment is composed of the same Vagrant Virtual Machines as in the testing environment. They are hosted in a private server that students do not have direct access to. The server is a 16-CPU machine based on the Intel(R) Xeon(R) CPU E5-2643 0 @ 3.30GHz. They two VMs will be used for the execution of the milestone and final submission, as well as for continuous feedback.

```
$ lscpu
Architecture:                   x86_64
CPU op-mode(s):                 32-bit, 64-bit
Byte Order:                     Little Endian
Address sizes:                  46 bits physical, 48 bits virtual
CPU(s):                         16
On-line CPU(s) list:            0-15
Thread(s) per core:             2
Core(s) per socket:             4
Socket(s):                      2
NUMA node(s):                   2
Vendor ID:                      GenuineIntel
CPU family:                     6
Model:                          45
Model name:                     Intel(R) Xeon(R) CPU E5-2643 0 @ 3.30GHz
Stepping:                       7
CPU MHz:                        1197.084
CPU max MHz:                    3500.0000
CPU min MHz:                    1200.0000
BogoMIPS:                       6583.96
Virtualization:                 VT-x
L1d cache:                      256 KiB
L1i cache:                      256 KiB
L2 cache:                       2 MiB
L3 cache:                       20 MiB
```

4.4.1 Execution of Milestone Submission

Shortly after the Milestone Submission deadline (see Section 1.1), your milestone submissions will be executed in the execution environment. For each group, scripts will automatically clone the group's repository and checkout the commit hash that the group has specified in the file `<group-name>-milestone.csv`. The submissions will be executed multiple times and in random order. The results will be published in a webpage[24]. For groups whose submission has errors feedback will be provided in the form of error messages.

Tip: Take advantage of these error messages to debug your server.

[24] `https://www2.compute.dtu.dk/courses/02159/os-challenge/milestone.html`

4.4.2 Continuous Feedback

Between the milestone and final deadline, the execution environment will operate in continuous feedback mode. Each night, it will pull and execute the latest commit in the default branch. The results will be published in a webpage[25]. Different to the execution of the milestone and final submission, the webpage will plot the best performance achieved. Thus, you shall use this to track the performance improvement of your server over time. For groups whose commit has errors feedback will be provided in the form of error messages.

Tip: Take advantage of these error messages to debug your server.

Important: The continuous feedback ranking is not the final ranking. Importantly, the final ranking will be executed with different random `seed`, and the ranking of groups with similar performance might depend on luck. It is recommended to experiment with multiple seeds and confirm that your solution does not overfit to a specific seed without generalising.

4.4.3 Execution of Final Submission

Shortly after the Final Submission deadline (see Section 1.1), your final submissions will be executed in the execution environment. For each group, scripts will automatically clone the group's repository and checkout the commit hash that the group has specified in the file `<group-name>-final.csv`. The submissions will be executed multiple times and in random order. The results and the winners will be presented at the end of the student presentation workshop.

4.5 Apple Silicon

The VMs of the execution environment are compatible with all major operating systems but require a machine with a x64 processor, including macOS machine with Intel processors. They are unfortunately not compatible with ARM-based processors, including macOS based on Apple Silicon.

For students who use a macOS machine with Apple Silicon (ARM64), two similar VMs are provided. These are specified in the `Vagrantfile` in the directory `arm64`[26]. At the time of writing, Virtualbox does not have stable support for ARM64, so we will use VMware instead. Note that VMware requires a licence key, but you can obtain one for non-commercial purposes for free[27]. In addition, you will need to install the Vagrant VMware Utility and the Vagrant VMware Desktop plugin. You can find installation guidelines here[28]. For reference, I can confirm that I have made this work in my laptop using the following software versions: Vagrant 2.3.7, VMware Fusion 13.0.2, Vagrant VMware Utility 1.0.22, Vagrant VMware Desktop Plugin 3.0.3. Once everything is installed correctly, the guidelines provided in Section 4.3 would still work with the exception that you would need to navigate to the `arm64` directory and use a different box.

```
$ cd arm64
$ vagrant box add starboard/ubuntu-arm64-20.04.5
```

For development, another option is to first develop the server for macOS and, once it compiles and runs without errors in macOS, you compile it and test it in Linux. Since both macOS and Linux use POSIX system calls, source code written for macOS should compile in Linux with minor or no adjustments. For convenience, the provided header file `message.h`[29] provides macros that convert the Linux endianess-related functions to macOS. This also serves as an example on how you can use macros to write source code that compiles in multiple operating systems. Binaries for macOS with ARM64 are also provided[30].

For testing and making sure that your server compiles and correctly runs in a Linux environment with a x64 processor, you can also use the GBAR Linux machines[31]. You can log in to two machines and use one for the client and one for the server. **Tip:** You can find the IP address of the server using the command `netstat -ie` and checking for the `eth0` interface. It is very important to avoid running very heavy workloads on the GBAR machines with are shared and used among all DTU students. To that end, limit your client configuration to low `difficulty` and make sure that your server returns within few minutes. Remember to also reduce the `delay` accordingly to

[25]`https://www2.compute.dtu.dk/courses/02159/os-challenge/continuous.html`
[26]`https://github.com/dtu-ese/os-challenge-common.git`
[27]`https://customerconnect.vmware.com/evalcenter?p=fusion-player-personal-13`
[28]`https://gist.github.com/sbailliez/2305d831ebcf56094fd432a8717bed93`
[29]`https://github.com/dtu-ese/os-challenge-common/blob/master/messages.h`
[30]`https://github.com/dtu-ese/os-challenge-common/tree/master/arm64/bin/macos`
[31]`https://www.gbar.dtu.dk/index.php/faq/53-ssh`

ensure that your server can handle multiple requests in parallel. If someone else in your group has a x64 machine, an additional solution is to use their machine for testing.

In any case, during the continuous feedback phase (see Section 4.4.2), you will be able to compile and run your server on the execution environment. Any errors or bugs not captured with the previous workarounds, should be captured there.

5 Submission Checklists

Here, you can find checklists with what you need to submit at each deadline.

5.1 Group Information File Submission

The group should:

- Identify all the group members;
- Select a group name;
- Create a private git repository;
- Generate a secret 6-digit random number;
- Create the group information file named `<group-name>.csv`; the file must be formatted exactly as specified in Section 3.2.

A representative of the group should submit the group information file.

5.2 Milestone Submission

The group should:

- Include the source code of the server and the `Makefile` in the git root directory;
- Confirm that the server compiles and runs in the testing environment;
- Confirm that the server yields 100% reliability;
- Give read permissions to the group's git repository to the teachers;
- Identify the git hash of the milestone commit.
- Create and submit the `<group-name>-milestone.csv` file.

A representative of the group should submit the milestone information file.

5.3 Final Submission

The group should:

- Include the source code of the server and the `Makefile` in the git root directory;
- Confirm that the server compiles and runs in the test environment;
- Confirm that the server yields 100% reliability;
- Include the code of each experiment in the repository (at least one experiment per group member);
- Comment the source code;
- Document the final solution and each experiment in the `README.md`;
- Confirm that the teachers have read permissions to the group's git repository;
- Identify the git hash of the final commit.

- Create and submit the `<group-name>-final.csv` file.
- Convert the `README.md` into a `pdf` file and submit it.

A representative of the group should submit the final information file and the report in `pdf` format.

5.4 Presentation

The group should:

- Prepare slides that present the final solution;
- Prepare slides that present the experiments.

6 Frequently Asked Questions

6.1 How should we document an experiment in the `README.md`?

An experiment compares two (or more) solutions to identify the best. An experiment description has 4 parts. Firstly, you should define and motivate the experiment. Then, you present the experiment setup, for example the configuration parameters of the client. Then, you present the results of the experiment either in the form of a table or a figure. Lastly, you discuss the results and derive into a conclusion regarding which of the solutions you adopt.

6.2 What do we need to do to win the challenge?

To win the challenge, your group should submit the solution that ranks first in the final submission execution. Your solution should compile and run in the execution environment, yield 100% reliability, as well as the lowest `score`. Your solution should also adhere to all the rules as specified in Section 2.6.

6.3 How will the project be assessed?

The final assessment for the project will be based on meeting learning objectives and not on the rankings. Each group member will be assessed **individually** in the scale: *Excellent*, *Good*, *OK*, *Poor* and *Fail*. The `score` and the ranking are irrelevant for the final grade. Instead, the motivation, design, implementation, and execution of the experiments carry most of the weight.

To receive *Good*, your group should submit a solution that compiles and runs in the execution environment, and yields 100% reliability. Your solution should also adhere to all the rules as specified in Section 2.6. Your code should be commented, and your repository should be well organised. The final solution should be documented in the `README.md` and presented at the final workshop. In addition, you, as an individual, should conduct at least one experiment. The experiment should be documented in the `README.md` and presented at the final workshop.

To receive *Excellent*, your individual performance should be above and beyond the requirements for *Good*. For example, conducting and documenting many experiments or conducting a particularly clever and challenging experiment are ways to demonstrate *Excellent* performance.

To receive *OK*, your individual performance should be below the requirements for *Good*. Essentially, you meet all the requirements for *Good*, but at a basic level (*e.g.* you did a very simple experiment) or you have some other imperfections (*e.g.* poorly written report).

To receive *Poor*, your individual performance should be below the requirements for *Good*. Essentially, you made a decent effort, but you miss some of the requirements for *Good*.

To receive *Fail*, your individual performance should be below the requirements for *Poor*. You miss many of the requirements for *Good* or you didn't submit a project at all.

6.4 What do I need to do to get 12?

The final grade depends on the overall assessment of the project and the final written exam. Getting *Good* or *Excellent* at the project will boost your grade upwards. Getting *Poor* or *Fail* at the project will push your grade downwards.

6.5 Is it possible to win the challenge and get less than 12?

Yes. Example: Alice's team won the challenge, but she did not conduct any experiments. She does not get a 12.

6.6 Is it possible to perform bad at the challenge but still get 12?

Yes. Example: Bob's team ranked last. Nevertheless, Bob executed a clever experiment. He documented it well and commented the source code. He also performed well in the written examination. Bob gets a 12.

6.7 The team feels too large for the milestone. How can we split the workload?

Organising the workload and learning how to work in large teams is part of the learning objectives of the project. It is important to understand that the development of the server is not just coding. The task needs developers, yes, but it also needs researchers (*i.e.* people who read, study, investigate, find development strategies), it needs testers (*i.e.* people who try out different client parameters and look for bugs), and it needs debuggers. You may also choose to begin with the experiments early on. Remember, I expect to see commits of code from all group members for the final submission only. It is OK if the milestone submission does not contain commits from every group member.

6.8 We want to compare 5 algorithms. Does this count as 1 or 5 experiments?

An experiment is defined as the comparison of one algorithm / implementation against the reference implementation (*e.g.* the milestone submission). Thus, this counts as 5 experiments. If your group concludes that there are 5 interesting ways to implement a particular part of the server, this can be naturally organised into 5 experiments distributed among 5 team members: each implements one algorithm and compares it against the benchmark.

6.9 How shall we best collaborate as a team?

Organising the workload and learning how to work in large teams is part of the learning objectives of the project. The key to success is good collaboration among the team members. It is natural that each group will have members that are more experienced with C Programming than others. Yet, all group members have a common goal and it is in the best interest of the group to use the common pool of resources in the best way. Remember that, similarly to Operating Systems, many working in parallel is always faster than one working sequentially!

A clever team member who is a beginner in C Programming will use his/her more experienced teammates as a way to learn more. Talk with them, study their code, try to understand what they do and why they do it. For example, trying to re-implement a basic server from scratch and compare it to their server, can be a very valuable learning experience. Collaborating with more advanced teammates is always a great opportunity for self-improvement.

A clever team member who is very experienced in C Programming will soon realise that working alone is limiting. One person alone can only do so many experiments, as we are all constrained by our own time limitations. Instead, if he/she invests some of this time to acting as a mentor or leader for the team members that are eager to learn from him/her, the invested time can be multiplied many times. Also, he/she will take ownership of the most difficult experiments, leaving the easier ones for the beginners, effectively making the best use of the common pool of resources.

There are multiple ways to collaborate as a team. I would personally opt for the following. The design of the experiments is first discussed and agreed at group level. In turn, each group member takes ownership of one experiment, implements it, and compares it against the benchmark. Finally, the results are discussed at group level and a conclusion is reached. For example, there are dozens of scheduling algorithms. A group discussion on their advantages and disadvantages may conclude on 5 candidates that are worth an experiment. The 5 algorithms are organised into 5 experiments and tested by 5 team members who work in parallel. Finally, the group discusses the results and selects the best.

6.10 I cannot find a group. Are teams of 1 person allowed?

It is recommended to participate in the challenge in a group of 2-4 persons. If, however, you learn better working alone or there are extenuating circumstances that make it difficult to work in a group, a group of 1 person is allowed.

6.11 Is it OK if our server works only for particular number of requests (`total`) specified in `run-client-final.sh`?

No, this is not OK. Your server should work for any number of requests (`total`). It is fine to be optimised to the configuration of `run-client-final.sh` (*i.e.* work faster than other configurations), but it should work reliably in any arbitrary reasonable client configuration.

6.12 I tried an online tool to get the hash of a number and I get a hash that is different to what the client sends for the same number. How is this possible?

SHA256 takes as input a byte array (not numbers or strings), so the hash you get from online tools depends on how the number is encoded by that tool at binary level. For example, 1 as a string will give you different hash than 1 as `uint64_t` little endian, and 1 as `uint8_t` will give different hash than 1 as `uint32_t`, etc. All these examples are different at binary level. If you'd like to use an online tool, it is important to pick one that allows you to specify the input in hex format, like `https://blockchain-academy.hs-mittweida.de/sha-256-generator/`. Then try out `01` (1 in `uint8_t`), `0001` (1 in `uint16_t` big endian), `0100` (1 in `uint16_t` little endian), etc. You'll see you get different output.

In our case, we want to feed the hashing function with an `uint64_t`, that is a byte array of 8 bytes. This means that you must provide these 8 bytes exactly the same way, i.e. little endian, as the client does to get the correct output.

6.13 Our server fails to handle many parallel requests. What can we do?

Answering this question is part of the challenge. However, one quick tip is to make sure that the bottleneck is not at the OS pending connections queue. For more information, check the `backlog` argument of the `listen()` system call.

6.14 VirtualBox fails to set the IP address of the VM. How can I fix this?

You can tell VirtualBox to allow additional IP ranges by configuring `/etc/vbox/networks.conf`. To allow the IP address the client and server uses you can use:

```
* 192.168.0.0/16
```

6.15 Can I use Windows to develop my server?

Your server must compile and run without errors in the execution environment that is Linux. Windows does not use POSIX system calls, therefore it is not straightforward to compile in Linux code written for Windows. There are some workarounds, like Cygwin[32] that address this issue, but it is recommended to compile your code inside the provided Linux VMs. You can still use the synchronised folders to write your source code in Windows and then compile it inside the VM.

6.16 I get errors when I try to add the vagrant box in Windows. What can I do?

These are likely due to firewall configurations or because your system does not trust the site that hosts the box. If you are facing this issue, reach out to the teacher as, thanks to a previous TA, we have a document with some solutions that might work for you.

7 Document Version History

- Version 2023-01:
  - Initial version.
- Version 2023-02:

[32]`https://www.cygwin.com/`

- Recommended Gitlab server changed to `gitlab.gbar.dtu.dk`.
- Typos fixed.