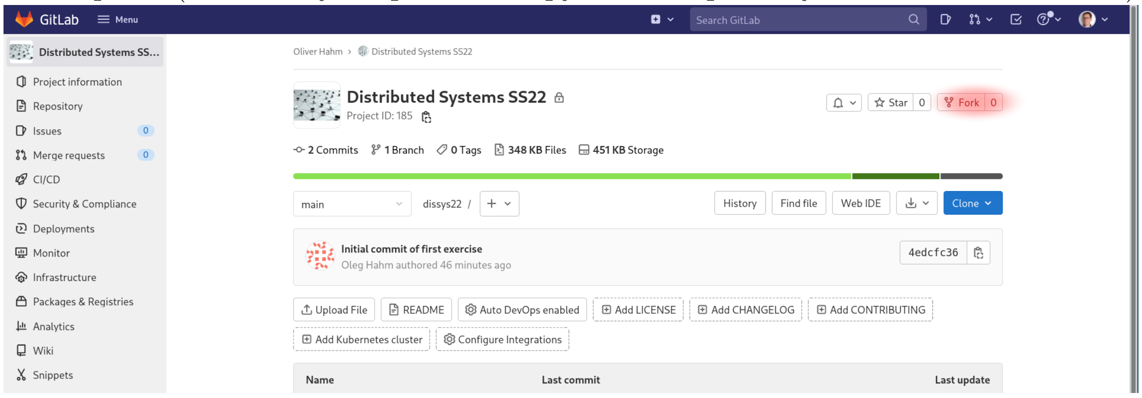
# Exercise Sheet 1

## Exercise 1 (Clone the Repository)

For the exercises of this course we will work with *git*. The submission for each exercise sheet must be committed to a *git* repository and pushed to a *remote*. In order to do so, you will first have to *fork* the course's main repository from the faculty's *GitLab* instance. The URL for this main repository is:

`https://gitlab.informatik.fb2.hs-intern.de/hahm/dissys22.git`

On the upper right corner you will find the fork button. Create a fork in your personal work space. (A fork is your personal copy of the repository on the GitLab server.)



Next you will have to *clone* your fork to your computer as a local repository. You can clone your fork from the command line by calling
`git clone <repo>`
(You find the URL for your fork on the GitLab page by clicking on the *Clone* button. Make sure that you are viewing your fork and **not** the original repository.)
Now click on *Project information* on the upper left and select *Members*. Click on the *Invite members* button and invite *Oliver Hahm* in the role *maintainer*.

Once you have successfully cloned your fork, you can start editing the files in your workspace. You can check for local modifications in your workspace by calling
`git diff`

In order to commit local changes to the repository **locally**, call
`git add <filename>` and
`git commit`
and set an appropriate *commit message*.

In order to push the local repository upstream to your fork, call
`git push origin main`
**ATTENTION: Do not forget this step once your solution is ready for submission! Otherwise your submission cannot be assessed by the lecturer.**

## Exercise 2 (Work with the Repository)

Open the the file `mybcp.c` in the editor of your choosing and modify the line

```c
printf("Hello world!\n");
```

into

```c
printf("Hello distributed systems!\n");
```

Save the modifications into the file. Check the local modifications (`git status` and `git diff`) before committing and pushing the changes to the upstream repository (`git add`, `git commit`, and `git push`).

## Exercise 3 (Programming C)

We will program in C in the exercises of this course. Even if you have not yet programmed in C, you will probably understand the basics of C rather quickly. You can find multiple books and online tutorials about programming in C, for instance,

- J. Gusted, *Modern C*: `https://modernc.gforge.inria.fr/`
- J. Wolf, *C von A bis Z*: `http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/`

It is expected that you will use a coding style which makes your code readable for other persons. This means in particlar that you will use consistent indentations and formatting of your code. It is preferable to put all blocks after `if/else/for/while/...` in curly brackets. You can separate lines that are not coherent in terms of content with blank lines. A useful style guide for readable code can be found, for instance, here:
`https://www.kernel.org/doc/Documentation/process/coding-style.rst`

## Exercise 4 (Error and return code handling)

Please note that library functions will report the status of the called operation via their return code. Typically a successful operation will return a zero or a positive number. Errors are usually reported by returning −1. Further error messages may be accessed via the system variable `errno` or by using the `perror` helper function:

```c
fd = open("filename", O_RDONLY);
if (fd == -1) {
    perror("Error on open");
    exit(EXIT_FAILURE);
}
```

Error handling and checking the return values is mandatory. You should always check them.

## Exercise 5 (Unbuffered file I/O)

In this exercise we will practice file handling via UNIX system calls. Note the necessary header files. System calls are documented via man pages in section 2 (other library functions can be found in section 3).

| | |
|---|---|
| `int open(const char *name, int oflag);`<br>`int open(const char *name, int oflag, mode_t mode);` | Open a file |
| `int creat(const char *name, mode_t mode);`<br>`int open(name, O_WRONLY\|O_CREAT\|O_TRUNC, mode)` | Create a file |
| `int close(int fd);` | Close a file descriptor |
| `ssize_t read(int fd, void *buf, size_t nbytes);` | Read from a file |
| `ssize_t write(int fd, const void *buf, size_t nbytes);` | Write into a file |
| `off_t lseek(int fd, off_t offset, int whence);` | Position in a file |

1. Implement a program called `mybcp` which copies an arbitrary file byte by byte. The name of the source file and the name of the destination file shall be passed via the command line, i.e., a call of the program looks like:
   `mybcp <source> <dest>`.
   The created destination file shall have **only** read and write permissions for the owner (`rw- -- --`).

2. Implement a program called `mybappend` which appends the content of a file byte by byte to another existing file. The name of the both files shall be passed via the command line once again.

3. Implement a program called `myrevbcp` that works similarly to `mybcp` but copies the bytes in reversed order (→ use `lseek()`).
   (**Hint:** You can check your implementation by reversing a file twice and check whether the final result matches the original file using the tool `diff`.)

4. Implement a programm called `mycp` that works similarly to `mybcp` but allows for setting the buffer size for the `read()` and `write()` system calls via the command line (→ `mycp <source> <dest> <buffersize>`).

**Hint:** You can check the results of your programs via the provided test scripts. The scripts are named after the respective program, e.g., `test_mybcp.sh` for `mybcp`. The following command let you build and test your program with a single line:

```
$ make && ./test_mybcp.sh
cc -std=c11 [...] -o mybcp mybcp.c
Copy 'file1' to 'file2'
---[Program output]--------------
Copy file bytewise...
--------------------------------

Compare file sizes: OK
Compare the content of the files: OK
Check file permissions: OK
```

## Exercise 6 (File attributes)

| | |
|---|---|
| `int stat(const char *name, struct stat *buf);`<br>`int fstat(int fd, struct stat *buf);` | Get file attributes |
| `int truncate(const char *name, off_t length);`<br>`int ftruncate(int fd, off_t length);` | Set the file size |

1. Implement a program called `filelength` that prints the size of a file. The name of the file is passed once again via the command line. Compare the output of your program with the output of `ls -l`.

2. Implement a program called `grow` which sets the size of a file to the given size. The name of the file and the size to be set are passed via the command line ($\rightarrow$ `grow <file> <size>`). If the file does not yet exist, it shall be created with read and write permissions for the owner.

3. Modify the program `mycp` in a way that the permissions for the original and the copied files are identical.

4. Modify the program `mycp` that it will print an error message when the source file is not a regular file.