University of Bamberg
Data Engineering
Prof. Dr. Maximilian E. Schüle

Exercise for *Systems Programming in C++* Winter Term 2023

Jakob Hornung (`mailto:jakob_hornung@stud.uni-bamberg.de`)
`https://vc.uni-bamberg.de/course/view.php?id=64576`
`https://vc.uni-bamberg.de/user/index.php?id=64574`

Sheet Nr. 01 (due on 31.10.2023 at 18:00)

Create a fork of the Git repository at `https://gitlab.rz.uni-bamberg.de/dt/sheet01`. The repository contains a subdirectory for each exercise on this sheet, into which you should put your answers. **Do not forget to `git push` your solution before the deadline expires.**

Once the deadline for this sheet expires, a signed tag will be created in your fork automatically. Your solutions will be graded based on the state of your repository at this tag. **Do not attempt to modify or remove this tag, as we cannot grade your solution otherwise.**

Exercise 1 (10 points)

On Linux, a plethora of useful information can be found in the system *manual pages (man pages)*. They are especially relevant to a systems programmer, who often needs to interact with the kernel through system calls.

While man pages can of course also be found on the internet, we prefer to use the `man` executable to view man pages offline. That way, we can be sure that we are viewing the version of a man page that matches the programs installed on our system.

(a) `man` does not display the text of the man page itself but it uses a helper program. Explain how `man` decides which helper program to use (Hint: Read the man page of `man`) in one sentence. How is the program called that is used on your system? Briefly describe your steps for finding the answer (1–2 sentences). List all commands you use.

Most systems usally use the program `less` to display man pages (although your system might be different). As man pages can be rather large, it is essential to be able to quickly search through man pages for relevant information.

(b) Check the man page of `less`. How can you quickly search for all occurrences of a keyword (name the command)? How can you jump to specific lines in a file (name the command)? Briefly describe your steps for finding the answer (1–2 sentences).

Exercise 2 (20 points)

The `proc` filesystem (also called `procfs`) provides an interface to kernel data structures which can be queried for information about your system.

(a) Use your knowledge about man pages and find a way to obtain information about your CPU with the `procfs`. Answer the following questions: Which file in the `procfs` contains the information about the available CPUs? Which helper program can be used to display the CPU infos in a nicer format?

In particular, the kernel exposes CPU flags which may (e.g., on ARM) influence the behavior of the `g++` option `-march=native` which was briefly introduced in the lecture. This option potentially sets a large number of other `-m<option>` options specific to the current CPU.

(b) Check the `g++` man page for information about the various `--help` options. Use this information to answer the following question: Which `-m<option>` options does `g++` set on your system when using `-march=native`? Briefly describe your steps for finding the answer (1 sentence, also list all commands that you used).

(c) Select 5 options that `g++` sets on your system when using `-march=native` from the last question. Briefly summarize their effect on the program that is produced by `g++` (one sentence per option).

(d) If portability is an issue, it is usually not desirable to blindly apply the `-march=native` option during compilation. Explain why this is the case, and briefly outline a more flexible approach which leverages a build system (<=3 sentences).

Exercise 3 (15 points)

In the lecture, the GNU `make` utility has been introduced.

(a) Briefly explain the purpose of `make` (1 sentence).

The documentation of GNU `make` can be found online (`https://www.gnu.org/software/make/manual/make.html`). Note that the `make` man page documents the usage of the actual `make` executable, while the online documentation is mostly about writing Makefiles.

(b) The most important parts of a Makefile are *rules* which contain *recipes*. Briefly explain these two concepts (1 sentence each). What is a *phony target*, and when do we want to use one (1 sentence)?

(c) A Makefile typically contains variables which control various aspects of the build process. Consult the online documentation and briefly describe the two *flavors* (*simply expanded* and *recursively expanded*) that variables can have (1 sentence each).

(d) Variables need to be set to values at some point, which is done with the `=`, `?=` and `+=` operators most of the time. Briefly explain when we would like to use which operator (1–2 sentences each). How can the value of a variable be influenced *without* modifying the Makefile (one sentence)?

(e) Briefly explain what *automatic variables* are (1 sentence). What is the purpose of the variables `$@`, `$<`, and `$^` (one sentence each)?

Exercise 4 (25 points)

In the lecture we introduced CMake — a build tool specialized for C and C++. CMake can not only be used to define executables and their dependencies, the language used for the `CMakeLists.txt` is essentially an entire programming language. It has control flow structures (`if`, `for`), etc. Also, it has several functions that can be used to extract information about the current system and its compiler. To allow easier customization for users that want to compile a project that uses CMake, it is possible to define *cache variables*. They can be changed by the user without having to modify the `CMakeLists.txt`. You can find more information about those concepts in the CMake documentation which you can find at `https://cmake.org/cmake/help/latest/`.

In this exercise we will focus on those aspects of CMake that are not directly related to source files, executables, libraries, etc.

(a) Read the CMake documentation to answer the following questions.

Which commands and/or variables can be used to

- Force CMake to compile C++ code using the C++20 standard?

Which commands and/or variables can be used to identify the following facts about the system:

- Is this system UNIX-like, macOS or Windows?
- Does the compiler support the flag `-march=native`?
- Does the system have the program `sed`?

For each question and fact in the list write down the commands and/or variables that can be used and explain each of them briefly (one sentence).

(b) Write a `CMakeLists.txt` that defines the two *cache variables* `ENABLE_NATIVE_ARCH` (default value: `OFF`) and `NATIVE_ARCH_FLAGS` (default value: empty string). When `ENABLE_NATIVE_ARCH` is set to `ON` by the user, your `CMakeLists.txt` should add a compiler flag to be used for all C++ files. `NATIVE_ARCH_FLAGS` is a list of flags that should be tried in the specified order.

Your `CMakeLists.txt` should determine the compiler flag that should be used (when `ENABLE_NATIVE_ARCH` is set to `ON`) as follows: First, all flags given by the user in `NATIVE_ARCH_FLAGS` should be tried. If none of them work (or if the list is empty), the following flags should be tried as a fallback in this order: `-march=native` (see task (a)), `-march=sandybridge`, `/arch:AVX`. The first flag that works should then be used for the compilation of all C++ files. If no working flag could be found, your `CMakeLists.txt` should output a warning message that tells the user that the `ENABLE_NATIVE_ARCH` option is not supported.

Exercise 5 (30 points)

In the lecture you learned several `git` commands. In this exercise you will have to use most of them to solve several problems in an existing Git project. All documentation about Git and its commands can be found in its man pages. This exercise makes use of the Git project that you can find at the following URL:

`https://gitlab.rz.uni-bamberg.de/dt/sheet01-git-exercise`

Before working on the tasks, first fork this repository by clicking on "Fork" in Gitlab. This will create a new copy of the repository owned by you. Work exclusively with your forked repository in this exercise.

Solve each of the following tasks by using Git commands. The Git commands should not only give the answer for this specific repository but be general enough to also work on another (similar) repository. In addition to the answers to the questions your solution should also **include all commands** you executed and briefly explain (1–2 sentences) what each command does.

(a) Clone the Git project. Which branches does the (remote) repository have? Which branch is checked out per default after cloning? Which tags does the (remote) repository have?

(b) Create a new commit on the `master` branch. It should add a new (empty) file with the name `testfile` and change the contents of the file `main.cpp` such that it says “Hi Earth!” instead of “Hello World!”. Then, push this commit to your Gitlab repository.

After task (b) the `master` branch should contain two commits.

(c) Now, switch to the branch `feature2`. Which commits does this branch contain that the `master` branch does not? Which commits does the `master` branch contain which `feature2` does not? Now merge the histories of both branches such that no existing commit is modified. The merged history should be visible in the `master` branch.

After task (c) the `master` branch should contain at least the two commits it contained after task (b) and the additional commits from the `feature2` branch.

In the next branch you will work on the branch `feature1`. This branch contains several commits that modify the file `main.cpp` and add a new file `test.cpp`. However, the changes are spread over several commits of which some do not contain useful commit messages. The goal for the next task is to clean up the commit history mainly by modifying existing commits. For this you will mainly need `git rebase` and/or `git reset`.

(d) Switch to the branch `feature1`. Modify the commit history so that User B ends up with exactly two commits in the end: One that contains only changes to `main.cpp` and another that contains only changes to `test.cpp`. Those commits should reuse the existing, meaningful commit messages but not the messages “Small fix” and “Fixes”. In the end, the new commit history should introduce the same changes as the old one. To verify this, find a command that outputs the summarized changes for this branch. Run this before and after modifying the history and compare its output.

After task (d) the branch `feature1` should contain exactly three commits.

(e) Try to push the changes to the branch `feature1` to your Gitlab repository. Explain why this fails (1 sentence). Find and run a command that allows you to push the changes safely. Choose a command that makes sure that changes that were introduced by others after you cloned the repository are not overwritten.

(f) Now merge the branch `feature1` with the `master` branch. Other than task (c) make sure that no merge commits are generated even if this means that the commit history of `feature1` is changed. Resolve all merge conflicts intelligently and describe what you did and why (one sentence). Make sure that the merged history is included into the `master` branch.

After task (f) the `master` branch should contain the same commits as after task (c) and additionally the two commits from the branch `feature1`.

(g) Finally, create the tag `v0.1` on the `master` branch. This tag should include the annotation “First release”. Push the tag to the Gitlab repository.