# Exercise Sheet 2 for Lecture Parallel Storage Systems

Deadline: 2024-05-07, 23:59

Prof. Dr. Michael Kuhn (michael.kuhn@ovgu.de)
Michael Blesel (michael.blesel@ovgu.de)
Parallel Computing and I/O • Institute for Intelligent Cooperating Systems
Faculty of Computer Science • Otto von Guericke University Magdeburg
https://parcio.ovgu.de

---

In this exercise sheet, you will practice the debugging of C applications with GDB and write your first application performing I/O.

## 1. Debugging C Applications (150 Bonus Points)

To be able to concentrate on efficient programming, we first want to look into the debugging of (parallel) applications. Therefore, we will work with the GNU debugger *GDB* and the memory checker *memcheck* from the *Valgrind* tool suite in this task. Both of these tools are usable for serial and parallel applications. Memory allocation is very error-prone in C and Valgrind can help detect common errors in memory management.

### First Steps

The `simple` directory contains a simple application that can be compiled with `make`.[1] This application will be used for you to familiarize yourself with GDB and Valgrind. It contains four functions, which return a pointer to an array or to a value inside an array and a `main` function that outputs the returned values. Unfortunately, this application contains multiple errors.

- Perform the following small tasks to get familiar with GDB. Document the used `gdb` commands and the output in a text file. A short GDB tutorial can for example be found at `https://www.cs.cmu.edu/~gilpin/tutorial/`.
  - Place a breakpoint on the `mistake1` function, start the application and print out the values of `buf` and `buf[2]`. Continue to the next line and repeat the printing of both values. What is the type of `buf`?
  - Place a breakpoint on the `mistake2` function and continue the application execution. What is the type of `buf`?
  - Continue the application execution. What output do you get now? Display the code around the current line. Which frames are on the stack? Switch to frame 1. Print out the contents of `p`.
  - Call the `mistake3` function in GDB (look up how to call functions directly).

[1] For an introduction into Makefiles, see `http://swcarpentry.github.io/make-novice/`.

- Firstly, modify the application so that it does not crash anymore. Try to keep the amount of code modifications to a minimum. Use `gdb` to find the errors in the code. The correct application output should look like this:

```
1 : 1
2 : 2
3 : 3
```

- Now the application is running, but it still contains memory errors that can occur (more or less at random) depending on the environment. Use Valgrind's `memcheck` tool to find the memory management errors and modify the application so that every function allocates memory correctly and that all memory is freed correctly before the termination of the application. Execute the program using `valgrind ./simple`.

  Note: Simply allocating the memory with `static` is *not* allowed. Also, global arrays and variables are not to be used. A short introduction to `valgrind` can for example be found at `https://www.valgrind.org/docs/manual/quick-start.html`.

Document the errors that cause application crashes and memory errors. For every error, state the corresponding lines of code that are erroneous and explain the reasons for the error (for example, multiple freeing of same memory).

# 2. Checkpoints

In the following task, you are supposed to make yourself familiar with the basic read and write operations in C. In the materials from the website you will find a `checkpoint` application. It contains a `calculate` function, which iteratively performs mathematical operations on a matrix. To execute the application, the number of threads and the iteration count have to be given as arguments.

In the output, you can see statistics about the application run like the used time, the throughput and the IOPS (I/O operations per second).

## 2.1. Writing Checkpoints (120 Points)

Modify the given application in a way that in every iteration a checkpoint file `matrix.out` is written. The writing process shall be done in parallel by all threads. The values of the matrix shall be overwritten in each iteration. To implement this look into the `write` and `pwrite` functions und consider the parallel environment (hint: thread-safety). Briefly describe how both functions work and explain your choice for this task. Integrate a header in the output file that contains the invocation parameters of the application:

- $T$: Threads
- $I$: Iterations
- $I_c$: The number of the last written iteration

Additionally, implement the calculations for the reached throughput and IOPS. Think about whether it is sensible to calculate the throughput and IOPS based on the total runtime or only based on the time needed for I/O operations. Give a short rationale for your chosen method.

### 2.2. Reading Checkpoints (60 Points)

The matrix is currently initialized with predefined static values. Implement the given `read_matrix` function. Extend the input parameters for this by adding a path to a `matrix.out` file. If no path is given or if the file does not exist the matrix shall be initialized statically as before. Think about appropriate error handling.

Lets look at two application runs $L_1$ and $L_2$ with their respective input parameters $T_1, I_1, T_2$ and $I_2$. $L_1$ has already finished and has written a checkpoint with the values for $T_1$, $I_1$ and $I_c$ in the header. Now $L_2$ shall be run. The checkpoint written by $L_1$ shall be the used as input by $L_2$. Initially the header shall be analyzed and the following scenarios have to be taken into account:

1. $T_1 \neq T_2$, where the matrix size is not dependent on the number of threads.
2. $I_c < I_1$
3. $I_c = I_1$ and $I_1 \geq I_2$
4. $I_c = I_1$ and $I_1 < I_2$

Think about and implement appropriate procedures for the first three mentioned cases. Provide explanations for your decisions.

For the last case, implement the reading of the values from the last written iteration. The current run ($L_2$) shall initialize the matrix with those values und continue the calculations from the next iteration ($I_c + 1$).

### 2.3. Atomic Checkpoints (60 Points)

Implement the atomic writing of checkpoints. This means that even in the case of a crash during checkpoint creation the checkpoint must still be in a consistent state. In this context consistent means that the written values are all from the same iteration and that $I_c$ shows the correct iteration number for these values. Discuss the pros and cons of two possible solutions for this task.

## Submission

We will count your last commit on the `main` branch of your repository before the exercise deadline as your submission. In the root directory of the repository, we expect a `PSS-2024-Exercise-02-Materials` directory with the following contents:

- A file `group.txt` with your group members (one per line) in the following format:

```
Erika Musterfrau <erika.musterfrau@example.com>

Max Mustermann <max.mustermann@example.com>
```

- A text file with the input and output for GDB called `gdb-output.txt`, a text file with error descriptions (reasons, code locations) called `simple-error.txt` and the modified source code in a directory called `simple` (Task 1)
- A text file with the answers for the questions called `checkpoint-answers.txt` and the modified source code in a directory called `checkpoint` (Task 2)