TDDD56 / TDDE65

Lab 0

Getting Started with C and Threads

Christoph Kessler

October 2025 / March 2026

Lab 0 (for TDDD56: on wednesday in the first course week, for TDDE65: on wednesday in the second course week) is **mandatory only for course participants who have *not* taken a course in concurrent programming and operating systems (or need to brush up C/thread programming skills)**.

We suggest that you do this lab together with a lab partner if any possible. It needs not be the same lab partner as during the actual lab series (TDDD56: CPU and GPU labs 1-6 in the subsequent weeks (46–51); TDDE65: Labs 1a, 1b, 2 and Miniproject).

There is no examination/demonstration for Lab 0. The main purpose is to identify potential needs for further reading-up on C and/or thread programming while there still is some time, and ultimately to give you a smoother start into the real lab series.

When the scheduled lab-0 time is up, continue with Tasks 1 and 2 on your own, so you finish this before Lab 1 starts.

Note: no supervision for Lab 0 is given after the scheduled Lab 0 session.

Preparation

BEFORE the lab session (i.e., before wednesday in the first week), work through the introductory slide set "Low-Level Programming in C and Multithreaded Programming", see the course web page. It focuses on those C constructs that are often perceived as difficult by programmers trained in Python or Java, such as pointers, type casts, storage classes, and memory management. If entirely new to C/C++, focus on this part first (needed for Task 1).

The slide set also gives an introduction to multithreaded programming with Pthreads (needed for Tasks 2 and 3).

Task 1: C Pointers and Memory

This lab assumes that you have worked through the C part of the slide set mentioned above.
Write a sequential C program that does the following:

1. Read in an integer number into a variable n.
2. Dynamically allocate an array `A` (as a local pointer variable in `main`) of `n` floatingpoint elements.
3. Calculate every array element `A[i] = i*i+1` for all `i` from `0` to `n-1`.
4. Measure the time taken for entire step 3. (The functions can be found on the lab page and an example is in the slide set.)
5. Output the result and the time.

   Additional experiments:

6. Also output the addresses of `i`, `A[0]`, `A[1]`, `A`, and `main`.
7. Use a global pointer variable for `A` instead and redo step 6.
8. Output the size (in bytes) of `i`, of `A[0]`, and of `A`.
9. What happens if you read from or write to `A` out of bounds?
10. Using suitable type casting, *convert* `A[1]` into an integer value and print it.
11. Using suitable type casting, *reinterpret* `A[1]` as an integer value and print it.

Questions

- How many bytes does a pointer variable have on your system?
- Why do we here need to use a pointer and dynamic memory allocation for `A` (step 2 resp. step 7), instead of simply defining a C array variable for it?

  How could this be changed if `n` were a compiler-known constant?

  Write a variant of this program to work with a constant `n` (and thus a fixed-size array).
- Run the entire program with execution time measurement (step 5) multiple times. How much variation of the time values do you observe across the runs? What could be the reason(s)?

Task 2: Multithreading with Pthreads

This task assumes that you have worked through the multithreading part of the introductory slide set mentioned above.

Extend the sequential C program from Task 1 to a multithreaded parallel program (using pthreads) that does the following (black text is from Task 1, blue text is new features added):

1. Read in an integer number into a variable `n`.
2. Dynamically allocate an array `A` of `n` floatingpoint elements.
3. Create multiple parallel pthreads, `p` in total including the initial thread, where `p` is a program parameter. Use only `p` $\leq$ number of performance-cores of the CPU (for the Olympen computers: 8).

   In parallel,

   calculate every array element `A[i] = i*i+1` for all `i` from `0` to `n-1`.

   (Every thread should take care of about an equal amount of elements so that the overall work is shared across all threads.)

   Then synchronize and shut down the created pthreads again (using `pthread_join`).
4. Measure the time taken for entire step 3.
5. Output the result (`A`, on one thread only) and the time.

Questions:

- Considering step 5 (output of all elements of A on one thread).

  Just as a thought experiment (no implementation necessary):

  What would/could happen if all threads output their share of elements in parallel, without further synchronization?
- Measure the execution time of step 3 for different values of p with fixed n.
- Compare the step-3 time of the multithreaded program (Task 2) with $p = 1$ threads (i.e., no further threads created) with the corresponding time of step 3 of the sequential program (Task 1).
- In general, parallel speedup over the sequential version by using more than one core could only be observed for very large n (better suppress outputting all elements then). Why is that?

Task 3 (as time permits)

Extend the multithreaded C program from Task 2 (black) as follows (new features are shown in blue color):

1. Read in an integer number into a variable `n`.
2. Dynamically allocate an array of `n` floatingpoint elements.

   Define a shared variable `sum` (to accumulate the global sum of all elements), initialized to 0.

   Create multiple (`p` in total) parallel pthreads, where `p` is a program parameter.

   In parallel,

   calculate every array element `A[i] = i*i+1` for all `i` from `0` to `n-1`.

   Every thread should take care of about an equal amount of elements so that the overall work is shared across all threads.

   In addition, calculate in parallel the global sum of all elements.

   (NB you may need synchronization here — why?)

   Then synchronize and shut down the created pthreads again (using `pthread_join`).

   Measure the time taken for entire step 3.

   Output the result, the global sum and the time.

Hint: Do the global-sum extension also for the sequential version (Task 1) for comparison.

Questions:

- Show by multiple example runs for fixed, sufficiently large n and $p > 1$ that, if not using synchronization, the global sum result may be wrong and the program is non-deterministic.
- Measure the step 3 time for fixed, sufficiently large n and for $p > 1$ (e.g., p=the number of (performance) cores of your system).
- Compare this time with a version of the program that does not use synchronization. Do you observe a slowdown when adding mutual-exclusion synchronization to the sum calculation in step 3? What could be the reason(s)?
- Try to eliminate most of this slowdown by using thread-local accumulator variables for the global sum computation.