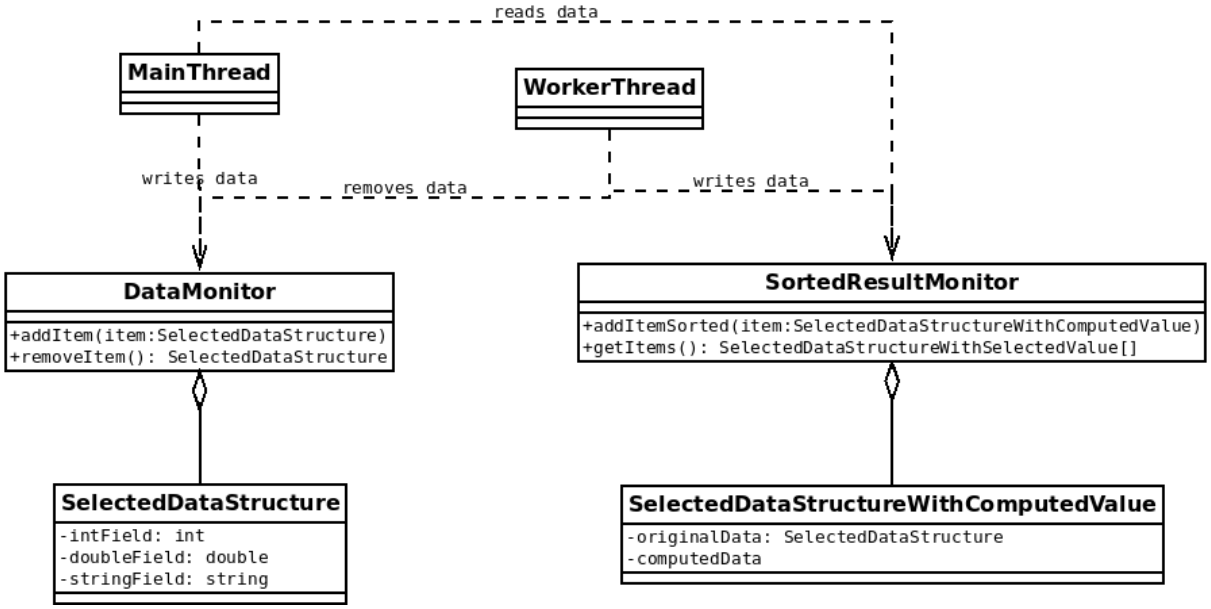
L1. Shared memory

Program a: application of consumer — producer pattern

TL;DR

Picture below shows a generalized view of how the program should work: two monitors are created — one for data and one for results. Each monitor contains an array, for unprocessed data and computed results respectively. Main thread reads data from a data file and writes them to the data monitor, while a selected amount of worker threads **concurrently** take (remove) items one by one from the data monitor, compute their result and write it to the result monitor. Program must handle cases when data monitor is empty and a worker tries to remove an item as well as when data monitor is full and main thread wants to add one more item – respective thread must wait until the size of the array changes. Goal of the main thread — copy all data read from the data file to the data monitor, wait for workers to finish and write all results from the result monitor to the result file. Goal of the workers — take (remove) item from data monitor, calculate result, decide if the result should go to the results and write it to result monitor if yes. Writing to result monitor should be implemented in such a way that it always remains sorted. Actions are repeated until all data are processed. **It is not required to use the structure shown in the diagram**.



Full description

Choose your own data structure that consists of 3 fields — one `string`, one `int` and one `double`, an operation that calculates a result from the data structure and a criterion to filter results by. Selected function should not be trivial so that it takes a bit of time to compute.

Prepare three data files containing **at least 25** elements of your selected structure each. Data file format is free to choose. It is **recommended** to use a standard data format (JSON, XML or other) and use existing libraries to read it. Name of the data file — **Group_LastnameF_L1_dat_x.json** (Group — your academic group, LastnameF — your last name and first letter of your first name, x — file number (3 files are required), file extension depends on your selected format). You have to prepare three data files:

- **Group_LastnameF_L1_dat_1.json** — all data matches your selected filter

criteria.

- **Group_LastnameF_L1_dat_2.json** — part of data matches your selected filter criteria.
- **Group_LastnameF_L1_dat_3.json** — no data matches your selected filter criteria.

Main thread works as follows:

1. Reads the data file to a local array, list or other data structure;
2. Spawns a selected amount of worker threads $2 \leq x \leq \frac{n}{4}$ (n — amount of data in the file).
3. Writes each read element to the data monitor. If the monitor is full, the thread is blocked until there is some free space.
4. Waits for all spawned worker thread to complete.
5. Writes all data from result monitor to the text result file as a table.

Worker threads work as follows:

- Take item from data monitor. If data monitor is empty, worker waits until there is data in the monitor.
- Computes the function selected by the student for the taken item.
- Checks if the result fits the selected criterion. If yes, the result is added to result monitor. The result monitor must stay sorted after insert.
- Work is repeated until all data from the file is processed.

Requirements for monitors

- It is recommended to implement the monitors as a class or a structure with at least two operations: to insert an item and to remove an item. If Rust is used, use Rust monitors.
- Data are stored in a fixed-size **array** (list or other data structure **is not allowed**).
- Data monitor array size cannot be larger than half of the elements in data file.
- Result monitor has enough size to contain all results.
- Operations with the monitor are protected using critical section where needed and thread blocking is implemented using conditional synchronization, using the tools of your selected language.
- It should be possible to fill and empty the data monitor (otherwise there is no point to have it :)).

Result file should be named **Group_LastnameF_L1_rez.txt**. File is formatted as a table with a header, filtered data with computed values are shown in the file.

Program b: manual data distribution between threads and OpenMP sum

TL;DR

Program does the same as program a, but OpenMP tools for critical section are used. Data monitor is not required, instead an algorithm to distribute data for the threads should be implemented, and each thread processes its part of the data. The parts should be as equal sized as possible. Additionally, compute sums of `int` and `float` fields using OpenMP tools and output and the end of the file.

Full description

Data file is the same as for program a, result file is the same but with an addition — sums of `int` and `float` fields have to be appended to the file.

Monitor has to be implemented using OpenMP synchronization tools (standard C++ tools cannot be used, they are used in program L1a).

The program has to spawn the same amount of threads and in program a, but the threads work directly with data array or vector. Each thread processes one part of the data. Data distribution has to be implemented by the student (not using OpenMP parallel loops). Elements have to be distributed as evenly as possible (if 27 elements are present in the data file and 6 threads are spawned, 3 threads process 4 elements each and 3 threads process 5 elements each).

Parallel region of the program computes the same function and applies the same filter as in program a with its part of the data. Sums of `int` and `float` fields have to be computed using OpenMP tools. Sums are computed only for the elements that match filter criteria. All computations are done in a single parallel region.

Lab programs:
a) C++, C#, Go, Rust — monitor is implemented using tools of chosen language;
b) C++ & OpenMP — monitor is implemented using tools of OpenMP;

L1 program grading

- L1a — 4 points;
- L1b — 2 points;
- Test — 4 points.

Deadline weeks for lab programs: a) 5, b) 6, test) 8.

Programs have to be presented during lab lectures not later than the deadline, program (.cpp, .cs, .go, .rs), data and result files (.txt) have to be uploaded to Moodle before presentation.

Recommended tools to parse JSON

- C++ — nlohmann

- C# — `DataContractJsonSerializer`
- Go — `Unmarshal`
- Rust — `serde_json`

Examples of data and result files

Student data is in the data file. **NB**: file should contain more elements than in the example; students should use their own data structure.

```
{
  "students": [
    {"name": "Antanas", "year": 1, "grade": 6.95},
    {"name": "Kazys", "year": 2, "grade": 8.65},
    {"name": "Petras", "year": 2, "grade": 7.01},
    {"name": "Sonata", "year": 3, "grade": 9.13},
    {"name": "Jonas", "year": 1, "grade": 6.95},
    {"name": "Martynas", "year": 3, "grade": 9.13},
    {"name": "Artūras", "year": 2, "grade": 7.01},
    {"name": "Vacys", "year": 2, "grade": 8.65},
    {"name": "Robertas", "year": 3, "grade": 6.43},
    {"name": "Mykolas", "year": 1, "grade": 6.95},
    {"name": "Aldona", "year": 3, "grade": 9.13},
    {"name": "Asta", "year": 2, "grade": 7.01},
    {"name": "Viktoras", "year": 2, "grade": 8.65},
    {"name": "Artūras", "year": 5, "grade": 8.32},
    {"name": "Vytas", "year": 3, "grade": 7.85},
    {"name": "Jonas", "year": 1, "grade": 6.95},
    {"name": "Zigmas", "year": 3, "grade": 9.13},
    {"name": "Artūras", "year": 2, "grade": 7.01},
    {"name": "Simas", "year": 3, "grade": 6.43}
  ]
}
```

Name, year and grade are concatenated into a single string and SHA1 hash is computed for it for each element in the data file. Only data with hashes that start with a letter (not a digit) are added to the result file. Hashes are sorted alphabetically.

Name	Year	Grade	Hash
Jonas	1	6,95	A18EAC8F30AC0FC630AE175A851CA5DA24FA8C85
Jonas	1	6,95	A18EAC8F30AC0FC630AE175A851CA5DA24FA8C85
Antanas	1	6,95	C4DE32954C067D224E22D55EB51F3774C3514B53
Asta	2	7,01	DC9556314AD1156B6E0CDBCB0927213B3316298D
Kazys	2	8,65	E763185F4B7303A787ACC513B7AA56706C7A42AC
Aldona	3	9,13	F1276E77671506983D8AE579D5BCC497F2F61346
Mykolas	1	6,95	FCBAB847E2FA87F8D54B77ED416B274D8677C61A

Creating OpenMP projects

Windows + Visual Studio

1. Open `Project` -> `Properties`.
2. Expand `Configuration Properties`.

3. Expand `C/C++`.

4. Select `Language`.

5. Enable option `OpenMP Support`.

CMake

Add command to `CMakeLists.txt` file: `set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++17 -fopenmp")`

Ubuntu + g++

Provide `-fopenmp` parameter when compiling, e. g., `g++ -c test.cpp -o test.o -fopenmp`.

Configuring Rust environment

1. Download RustUp from https://rustup.rs/.

2. Run downloaded file, choose default options. The tool will configure the compiler.

3. Run GoLand environment, select Plugins, find Rust plugin in the search and install it. ToML plugin will be installed as well.

4. Restart GoLand, select New Project and then Rust. When creating the project, GoLand will suggest downloading Rust standard library if that has not yet been done. It is recommended to do it.

5. If a debugger is required, try running the Debug configuration and GoLand will offer intalling Native Debugging plugin. When it is installed, all GoLand's debugging tools will work with Rust projects.