Parallel and Distributed Computing (B4B36PDV)

Matěj Kafka, Michal Jakob

kafkamat@fel.cvut.cz

https://pdv.pages.fel.cvut.cz

Introduction to C++

C, but easier to use, and much more complex

- Started in 1982 as a superset of C.
  - Still mostly compatible with C libraries.
- Provides abstractions such as classes, generics,...
- Still oriented on low-level and high-performance computing.
  - "zero-cost abstractions" (unlike e.g. Java)
- Sprawling, complex language, multiple generations of overlapping features.
  - But handles a lot annoying issues that we would have with C.
- We will be using a carefully selected subset of the language.

Don't be afraid to ask questions about C++ at any time.

Coding session 1

BASICS OF C++

Multithreading in C++

- C++ provides cross-platform APIs for working with threads, synchronization primitives and atomics.
- Internally, these APIs typically wrap existing APIs provided by the platform (pthread on POSIX, Win32 on Windows,…).

- `std::jthread` (C++20)
- `std::mutex`, `std::unique_lock`, `std::scoped_lock`
- `std::condition_variable`

Multithreading with pthread

```
void* fn(void* arg) {
    ...
}

int main() {
    pthread_t thread;
    if (0 > pthread_create(&thread, nullptr, fn, nullptr)) {
        return 1;
    }

    ...
}
```

How to use pthread threads?

Multithreading with pthread

```
void* fn(void* arg) {
    ...
}

int main() {
    pthread_t thread;
    if (0 > pthread_create(&thread, nullptr, fn, nullptr)) {
        return 1;
    }

    ...

    void* return_value;
    if (0 > pthread_join(thread, &return_value)) {
        return 2;
    }
}
```

Coding session 2

MULTITHREADING IN C++

Preventing misunderstandings with the compiler and the CPU

ATOMIC OPERATIONS

Atomic operations

Why do we even need them?

```
uint32_t shared_var = 0;

void thread_fn() {
  for (size_t i = 0; i < 1'000'000; i++) {
    shared_var++;
  }
};

auto t1 = std::jthread(thread_fn);
auto t2 = std::jthread(thread_fn);
```

What values can shared_var have here?

Coding session 3

ATOMIC OPERATIONS

Atomic operations

- `std::atomic<T>`
- "Type that only lives in memory."

- What does it give us?

1. Ensures that operations on the value are not fragmented.
2. Ensures that operations are not implemented using multiple non-atomic operations (e.g., load-modify-store).
3. ... is that enough?

Atomic operations

Is it enough?

```cpp
bool value = false;
bool value_was_updated = false;

auto t1 = std::jthread([&] {
    // we set `value` to true
    value = true;
    // let the main thread know that we updated the value
    value_was_updated = true;
});

// here, we wait until the flag is updated
while (!value_was_updated) {}

// now, `value` must obviously be true
std::cout << "value = " << value << "\n";
```

Atomic operations

And now?

```cpp
bool value = false;
std::atomic<bool> value_was_updated = false;

auto t1 = std::jthread([&] {
    // we set `value` to true
    value = true;
    // let the main thread know that we updated the value
    value_was_updated = true;
});

// here, we wait until the flag is updated
while (!value_was_updated) {}

// now, `value` must obviously be true
std::cout << "value = " << value << "\n";
```

* detailed explanation on the last slide

Atomic operations
Memory order

- `std::atomic<T>`
- "Type that only lives in memory."

- What does it give us?

1. Ensures that operations on the value are not fragmented.
2. Ensures that operations are not implemented using multiple non-atomic operations (e.g., load-modify-store).
3. **Ensures causality between operations on different threads** (if we request it to do so = memory order).

Detailed explanation of the last example for atomic variables

Both the compiler and the CPU are allowed to re-order operations under the "as-if" rule – as long as the "observable behavior" of the code on a single thread is preserved, both are essentially allowed to make any optimizations. Notably, this means that the observed behavior **from another thread** can differ between CPUs and compilers.

For compilers, the input is your C++ program, and the output is the machine code, composed of CPU instructions. The compiler is allowed to reorder or even remove operations, as long as the end result is the same **as seen by the current thread**. For example, in the following program, compiler can increment `j` before `i`, because you won't be able to observe any difference in behavior:

```
uint32_t i = 0, j = 0;
...
i++;
j++;
```

However, in the following program, the reordering cannot be done, because it would change the behavior:

```
uint32_t i = 0, j = 0;
...
i++;
j += i;
```

Note that "observable behavior" is defined in the C++ standard in a specific way, but mostly aligns with developer intuition. After the compiler is done with the program, it is executed by the CPU, which also attempts to make it run faster. One of the optimizations is that the CPU effectively builds a dependency graph of the executed CPU instructions, and executes many instructions in parallel, as long as all their inputs are available ("superscalar CPU").

One of the outcomes is that some CPUs can also reorder memory loads and writes, **as long as they can "fake" the correct values for other instructions on the same core**. In practice, the CPU has a buffer where it stores pending write operations, and if a later instruction on the same CPU core reads from an address which has a pending write, the CPU forwards the value to the instruction to preserve the illusion of sequential execution, **before that value is actually written to the cache/RAM** and visible to other cores (threads).

As a result, it is possible that different cores will observe the **same writes in a different order**. Specifically in the example on the slide, if we did not use an atomic variable, we would risk that the CPU reorders the write to `value` after the write to `value_was_updated`.

When we use an atomic store for `value_was_updated` (instead of a normal one), it ensures **causality** between operations done on the spawned thread before the store, and operations after the read in the main thread. That is, if main thread observes the write to `value_was_updated`, it is guaranteed that the previous write to `value` is also visible, even though it is not an atomic operation. In practice, the atomic store prevents reordering any previous operation after it, and the atomic load prevents reordering any operations done after it before the load.