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

Sharing memory I

```
uint32_t shared_var = 0;

void thread1() {
    shared_var = 0x12345678;
}
void thread2() {
    std::cout << shared_var << "\n";
}

auto t1 = std::jthread(thread1);
auto t2 = std::jthread(thread2);
```

What values can `thread2` print?

Sharing memory I

Reads and writes might be fragmented!

```
uint32_t shared_var = 0;

void thread1() {
    shared_var = 0x5678;
    *(&shared_var+2) = 0x1234;
}
void thread2() {
    std::cout << shared_var << "\n";
}

auto t1 = std::jthread(thread1);
auto t2 = std::jthread(thread2);
```

Sharing memory II

```
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?

Sharing memory II

Increment might not be atomic!

```
uint32_t shared_var = 0;

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

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

Coding session 3

ATOMIC OPERATIONS

Atomic operations in C++

- `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. Prevents the compiler from eliding memory operations.
4. … is that enough? *

* continued in 4th lecture