CZECH TECHNICAL UNIVERSITY IN PRAGUE

# Parallel and Distributed Computing (B4B36PDV)


**Matěj Kafka,** Michal Jakob

kafkamat@fel.cvut.cz

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

# Shared data structures

How to share data between threads?

- Don't.
- Not always possible.
  - databases, graph algorithms,…
- Just put the data behind a mutex, right?
  - Fine if the data structure is not in the hot path.
  - Easily can become a bottleneck.

# Reader-writer lock

## Mutex that allows multiple readers

- Typically, it is safe to read data from multiple threads.
- But it's not safe to read+write or write+write.

- It's very common to read a lot but write rarely.
- We want to allow multiple readers XOR a single writer.

- std::shared_mutex

Coding session 1

# RW LOCK

# Reader-writer lock

## Mutex that allows multiple readers

- Typically, it is safe to read data from multiple threads.
- But it's not safe to read+write or write+write.

- It's very common to read a lot but write rarely.
- We want to allow multiple readers XOR a single writer.

- std::shared_mutex

- Higher overhead compared to a mutex.
- Possible writer starvation if there are many readers.

# Concurrent data structures

- Break up the data structure into smaller parts.
- Synchronize each part separately.
  - Multiple threads can operate on different parts of the DS without blocking.

Coding session 2

# CONCURRENT HASH SET

# Concurrent data structures

- Break up the data structure into smaller parts.
- Synchronize each part separately.
  - Multiple threads can operate on different parts of the DS without blocking.

- Sometimes we need to synchronize on the whole DS.
- Typically much faster than locking the whole DS.
- Granularity trade-off
  - The more locks we have, the higher the total overhead.
  - Sometimes we cannot find a good compromise.

# Lock-free data structures

## Remove all software locking

- Instead of using software mutexes, we can let hardware take care of synchronization -> atomic variables.

- What's a mutex anyway?

- Let us take a small detour…

# Atomic variables II

It does not get easier...

- `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?

Coding session 3

# ATOMIC VARIABLES

# Atomic variables II

## How about more complex operations?

- Typically, we need to do a more complex operation.
- Often, we can express the operation in three steps:
  1. Read the current state from the data structure.
  2. Do some thread-local operation (e.g. allocate a node, compare an existing value).
  3. Update the data structure with a new value.
- Read-Modify-Write (RMW)
  - You might know the concept from database transactions.

Idea: If we could do the update using a single atomic operation, we'd be much closer to solving the issue.

# Compare-and-swap

"CAS"

- We have a race condition between the read (step 1) and the update (step 3).
- Instead, we can make the update step conditional.

"If the previously read value is still the same, replace it with this new value."

```
// SINGLE ATOMIC OPERATION
if (*value == previous_value) {
    *value = new_value;
} else {
    previous_value = *value;
}
```

Coding session 4

# COMPARE AND SWAP

# Lock-free data structures

## A small disclaimer

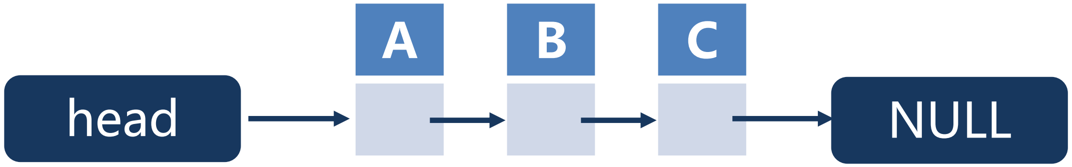
Do try this at home.

But (almost) never try it at work!

https://abseil.io/docs/cpp/atomic_danger

# Lock-free stack

- Basic data structure based on a linked list.
- 3 operations:
  - Push
  - Find
  - Pop




```
class node {
public:
    std::atomic<node*> m_next = nullptr;
    T m_value;

    explicit node(T value)
        : m_value(value) {}
};
```

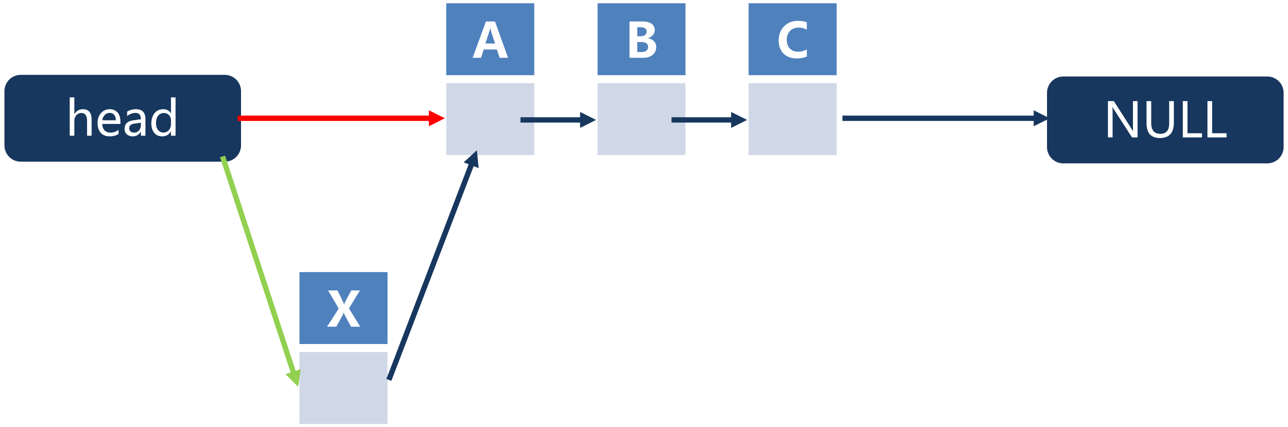
# Lock-free stack

Push (add new value to the top of the stack)

# Lock-free stack

Push (add new value to the top of the stack)

```
void push(T value) {
    auto new_node = new node(value);
    auto first_node = m_head.load();
    do {
        new_node->m_next = first_node;
        // if the condition below fails,
        // first_node` is updated to the current value
    } while (!m_head.compare_exchange_weak(first_node, new_node));
}
```



# Lock-free stack

## Find (does the stack contain a specific value?)

# Lock-free stack

Find (does the stack contain a specific value?)

```
bool contains(T value) const {
    auto node = m_head.load();
    while (node != nullptr) {
        if (node->m_value == value) {
            return true;
        }
        node = node->m_next.load();
    }
    return false;
}
```

# Lock-free stack

Pop (remove value from the top of the stack)

# Lock-free stack

Pop (remove value from the top of the stack)

```
// INCORRECT
std::optional<T> pop() {
    auto first_node = m_head.load();
    while (first_node != nullptr) {
        auto second_node = first_node->m_next.load();
        if (m_head.compare_exchange_weak(first_node, second_node))
        {
            auto value = first_node->m_value;
            delete first_node;
            return value;
        }
        // retry, first_node is updated to the new value
    }
    return {};
}
```

# ABA problem

- If I load an atomic value twice, and the value is still the same, it does not necessarily mean that it did not change in the meantime.
- If someone deallocates a node and then allocates a new one, allocator will often return the just-freed allocation.
- We need to augment the data to unambiguously know if there was a change since the last read.
  - Include a counter next to the pointer.
  - Note that the counter must be a part of the pointer (e.g. HEAD), not the target node.

https://en.wikipedia.org/wiki/ABA_problem#Examples

# Node deallocation

- We cannot touch the first node, since it may have been deallocated in the meantime.
- Not a problem in garbage-collected language.
- Very hard to solve in C++ (out of scope for PDV).
  - hazard pointers
  - RCU