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`

- Slightly 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.
- Often much higher throughput 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.
- Sometimes we need to synchronize on the whole DS.

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

- `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.

Coding session 3

ATOMIC VARIABLES

Atomic `max(a, b)`

read – modify – write

1. Read current maximum.
2. Compare with the local value.
3. If local value is greater, update the maximum.
   - Race condition!
   - We must prevent other threads from changing it.
   - But that's what we wanted to avoid in the first place.

Idea: Instead of preventing changes, how about detecting them and retrying?

Compare-and-swap

"CAS" (also known as compare_exchange)

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

"If the variable still has this value that we previously read, replace it with this new value."

```
// single atomic operation
if (*value == previous_value) {
    *value = new_value;
} else {
    previous_value = *value;
}
```

Compare-and-swap

"CAS" (also known as compare_exchange)

- If compare-and-swap fails, we need to roll back the changes and retry.

```
auto var = std::atomic<T>(...);

auto old = var.load();
while (true) {
    auto new = ... old ...;
    if (var.compare_exchange_weak(old, new)) {
        break;
    }
}
```

Coding session 4

COMPARE AND SWAP

Compare-and-swap

read – modify – write

- General pattern to implement many atomic operations.
- Express the operation as 3 steps:
  1. Atomically read current version of the shared data.
  2. Modify the local copy.
     - Must not be visible to others.
     - Must be able to roll back any changes.
  3. Write back the new value…
     - …while checking that there were no new changes.
     - On failure, roll back changes from 2. and retry from 1.
- Common even outside multithreaded programs:
  - database transactions
  - HTTP `ETag` and `If-Match` headers
  - `git pull -> git commit -> git push`

Atomic variables II

- `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?

Atomic operations

Is our definition sufficient?

```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. Prevents the compiler from eliding memory operations.
4. **Prevents the compiler and CPU from reordering other operations** (if we ask them to do so = "memory order").

Coding session 5

SPINLOCK

Lock-free data structures

A small disclaimer

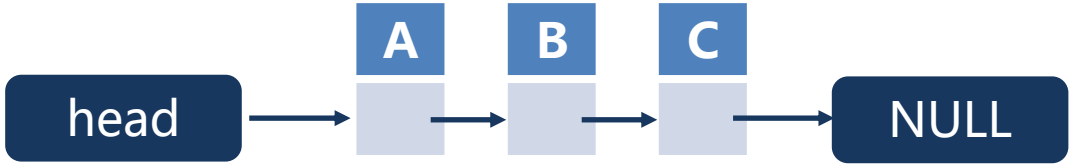
Do try this at home.

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.
  - Also useful as a building block for, e.g., lock-free hashmaps.
- 3 operations:
  - Push
  - Find
  - Pop




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

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

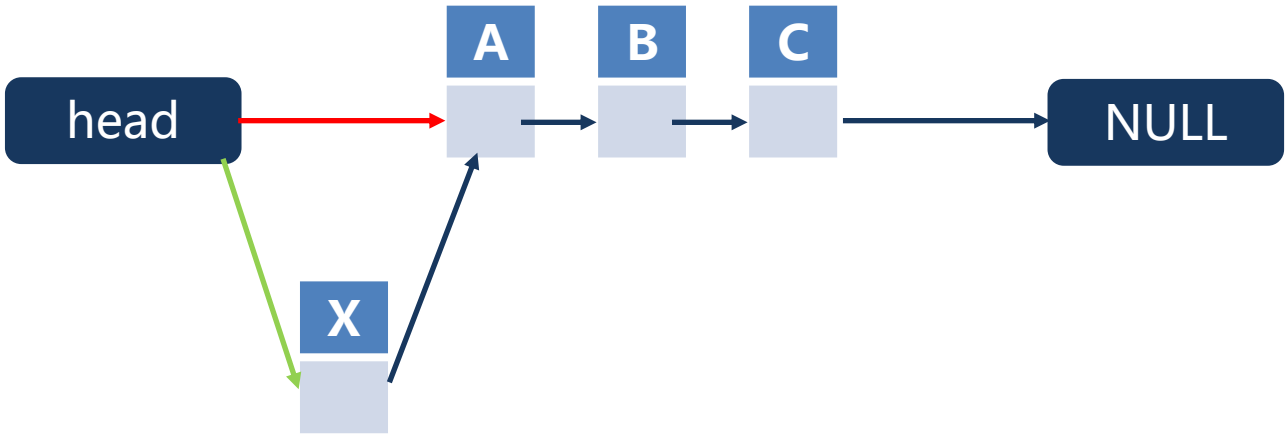
Lock-free stack

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));
}
```



* Not yet compatible with deletion, see later slides.

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;
}
```

* Not yet compatible with deletion, see later slides.

Lock-free stack

Pop (remove value from the top of the stack)

Lock-free stack

Pop* (remove value from the top of the stack)

```
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
    }
    // stack is empty
    return {};
}
```

* Not yet correct, see the following slides.

Node deallocation

How to free the node memory?

- All operations so far assumed that once we have a pointer to a node, we can safely access its fields.
- However, with deallocation, **we cannot safely touch anything inside the node**, since it may have been deallocated in the meantime.
- Note that this is not a problem in garbage-collected languages – since our thread holds a reference to the node, it will not be freed.
- Very hard to solve in C++ (out of scope for PDV).
  - Common solutions: hazard pointers, RCU
  - Generally, the data structure reimplements some form of local garbage collection, as the nodes must only be freed dynamically after all threads leave them.

Node deallocation

Do we need to free memory?

- Does this mean that the data structure is useless? No!
- Even insertion and lookup without deletion is often useful.
  - Surprisingly often, we just insert values and look them up, and only free the whole data structure at the end of the computation.
  - e.g., many databases, deduplicating values using a hashset, building a list of values to process in the next iteration,...
- We can occasionally synchronize all threads (e.g., using a barrier) and deallocate all unused nodes at once.
  - If we don't synchronize too often, we still retain concurrent access for most operations.
  - The program must have "safe points" where we can synchronize all threads (e.g., between iterations).

Reusing nodes

Avoiding wasted memory without deallocation.

- Another option: If we cannot deallocate the node, could we at least reuse it next time our thread needs to insert a new value?
- Effectively, we will create a custom allocator just for our nodes.
- Since the content of the memory is always a valid node, even if possibly unused, we can safely access any node pointer (if we're careful about our assumptions).
- That way, we're not actually leaking much memory, unless we free a lot of nodes and never reuse them.

ABA problem

Look twice, see no change…

- If I load an atomic value twice, and get the same result, it does not imply there was no change in the meantime.
- If a thread pops a node and then inserts a new one, it may be "the same node" (have the same address), even though the content changed.
  - E.g., if we wanted to keep the stack sorted, this could result in inserting a value in the wrong place.
- To fix this, we can augment all pointers with a counter that is incremented on each change.
  - As a result, the CAS operation will fail after a change.
  - 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

Detailed explanation of memory order for atomic operations:

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.