The Two Flavors of Transactional Memory

This lesson explains synchronized and atomic blocks in transactional memory.

WE'LL COVER THE FOLLOWING

- Synchronized & Atomic Blocks
- Synchronized Blocks
- Atomic Blocks
- `transaction_safe` versus `transaction_unsafe` Code

> C++ supports transactional memory in two flavors: synchronized blocks and atomic blocks.

Synchronized & Atomic Blocks

Up to now, I only wrote about transactions. Now, I will write about synchronized blocks and atomic blocks. Both can be encapsulated in each other; specifically, synchronized blocks are not atomic blocks because they can execute transaction-unsafe. An example would be code like the output to the console which can not be undone. For this reason, synchronized blocks are often called relaxed blocks.

Synchronized Blocks

Synchronized blocks behave like they are protected by a global lock, i.e. This means that all synchronized blocks follow a total order. In particular: all changes to a synchronized block are available in the next synchronized block. There is a *synchronizes-with* relation between the synchronized blocks because of the commit of the transaction *synchronizes-with* the next start of a transaction. Synchronized blocks cannot cause a deadlock because they create a total order. While a classical lock protects a memory area, a global lock of a synchronized block protects the total program. This is the reason the following program is *well-defined*:

```
// synchronized.cpp

#include <iostream>
#include <vector>
#include <thread>

int i= 0;

void increment(){
  synchronized{
    std::cout << ++i << " ,";
  }
}

int main(){

  std::cout << std::endl;

  std::vector<std::thread> vecSyn(10);
  for(auto& thr: vecSyn)
    thr = std::thread([]{ for(int n = 0; n < 10; ++n) increment(); });
  for(auto& thr: vecSyn) thr.join();

  std::cout << "\n\n";

}
```

Although the variable `i` in line 7 is a global variable and the operations in the synchronized block are transaction-unsafe, the program is well-defined. 10 threads concurrently invoke the function `increment` (line 21) ten times, incrementing the variable `i` in line 11. The access to `i` and `std::cout` happens in total order. This is the characteristic of the synchronized block. Afterwards, the program returns the expected result; the values for `i` are written in an increasing sequence, separated by a comma.

What about data races? You can have them with synchronized blocks. A small modification of the source code is sufficient to introduce a data race

```
// nonsynchronized.cpp

#include <chrono>
#include <iostream>
#include <vector>
```

```
#include <thread>

using namespace std::chrono_literals;

using namespace std;

int i= 0;

void increment(){
  synchronized{
    cout << ++i << " ,";
    this_thread::sleep_for(1ns);
  }
}

int main(){

  cout << endl;

  vector<thread> vecSyn(10);
  vector<thread> vecUnsyn(10);

  for(auto& thr: vecSyn)
    thr = thread([]{ for(int n = 0; n < 10; ++n) increment(); });
  for(auto& thr: vecUnsyn)
    thr = thread([]{ for(int n = 0; n < 10; ++n) cout << ++i << " ,"; });

  for(auto& thr: vecSyn) thr.join();
  for(auto& thr: vecUnsyn) thr.join();

  cout << "\n\n";

}
```

To observe the data race, I let the synchronized block sleep for a nanosecond (line 16). At the same time I access the output stream `std::cout` without a synchronized block (line 30). In total, 20 threads increment the global variable `i` - half of them without synchronization. The C++11 standard guarantees that the characters will be written atomically; that is not an issue. What is worse is that the variable `i` is written by at least 2 threads. This is a data race, hence, the program has undefined behavior. The total order of synchronized blocks also holds for atomic blocks.

Atomic Blocks

You can execute transaction-unsafe code in a synchronized block, but not in an atomic block. Atomic blocks are available in three forms: `atomic_noexcept`, `atomic_commit`, and `atomic_cancel`. The three suffixes `_noexcept`, `_commit`, and `_cancel` define how an atomic block should manage an exception.

atomic_noexcept: If an exception is thrown, `std::abort` will be called and the program aborts.

atomic_cancel: In the default case, `std::abort` is called. This will not hold if a transaction-safe exception is thrown that is responsible for ending the transaction. In this case the transaction will be canceled, put to its initial state, and the exception will be thrown.

atomic_commit: If an exception is thrown, the transaction will be committed.

Transaction-safe exceptions are: `std::bad_alloc`, `std::bad_array_length`, `std::bad_array_new_length`, `std::bad_cast`, `std::bad_typeid`, `std::bad_exception`, `std::exception`, and all exceptions are derived from one of these.

`transaction_safe` versus `transaction_unsafe` Code

You can declare a function as `transaction_safe` or attach the `transaction_unsafe` attribute to it.

```
int transactionSafeFunction() transaction_safe;

[[transaction_unsafe]] int transactionUnsafeFunction();
```

`transaction_safe` belongs to the type of the function. What does `transaction_safe` mean? A `transaction_safe` function is, according to the proposal N4265, a function that has a `transaction_safe` definition. This holds true if the following properties *do not* apply to its definition:

- It has a `volatile` parameter or a `volatile` variable.
- It has `transaction-unsafe` statements.
- If the function uses a constructor or destructor of a class in its body that has a `volatile` non-static member.

Of course this definition of `transaction_safe` is not sufficient because it uses the term `transaction_unsafe`. You can read the proposal N4265 for the details.