Threads II: Atomic Operations

PB173 Programming in Modern C++

Nikola Beneš, Vladimír Štill, Jiří Weiser

Faculty of Informatics, Masaryk University

spring 2016

Outline

- atomic operations
- memory barriers
- libaray: `std::atomic`, `std::atomic_flag`
- lock-free programming

Atomic Operations

“A need to execute an operation containing more CPU instructions.”

- use a mutex to guard to shared piece of code
  - (could be) expensive
    - *(depends on the implementation)*
  - context switching

Atomic Operations

How can we implement a mutex?

Atomic Operations

How can we implement a mutex?
Through the system call.

Atomic Operations

How can we implement a mutex?
Through the system call.
Oh, wait... How can we implement this feature in the OS?

Atomic Operations

How can we implement a mutex?
Through the system call.
Oh, wait... How can we implement this feature in the OS?

- parallel architectures have to provide low level synchronization primitives
  - special instructions in the native assembler
  - (sometimes) adopted by higher languages
- pre-C++11: only compiler-specific interface to those primitives
- C++11 standard defines common interface across platforms

Memory Barriers

Special (micro) instructions preventing both compiler and processor from reordering memory accesses.

- function calls to different compilation unit prevents compiler to reorder reads and writes
- `volatile` modifier prevents compiler from reordering accesses to `volatile` objects relatively to each other
- two different approaches to memory barriers
  - acquire semantics
  - release semantics

Memory Barriers
Acquire Semantics

*Read access tagged with acquire semantics causes that no other read operation placed **after** the tagged access can be executed before the tagged access.*

The access to `sharedZ` variable cannot occur before the access to `sharedY` variable.

```
int x = sharedX;
int y = sharedY; // this is tagged access
int z = sharedZ;
```

Memory Barriers

Release Semantics

> *Write access tagged with release semantics causes that no other write operation places **before** the tagged access can be executed after the tagged access. By the time of tagged accessing, every write access which happened before is visible.*

The access to `sharedX` variable cannot occur after the access to `sharedY` variable. Any other thread can see new values in both `sharedX` and `sharedY`.

```
sharedX = x;
sharedY = y; // this is tagged access
sharedZ = z;
```

Memory Barriers

...and Mutexes

- mutex lock
  - acquire semantics
- mutex unlock
  - release semantics
- wait on conditional variable
  - both acquire and release semantics
- notify on conditional variable
  - C++ standard does not specify
  - POSIX says it has release semantics

Memory Ordering in C++11

for barriers and atomic operations

- relaxed
  - no ordering, just atomic operation
- acquire
- release
- release-acquire
  - combines together
  - for compound operations (increment, exchange)
- sequence semantics
  - release-acquire + total ordering
  - default ordering
  - **recommended approach**

STD – Atomic

`std::atomic_flag`

- standard guarantees atomicity
- two operations
  - test and set
    - assings true, returns the previous value
  - reset
    - assigns false

`std::atomic<T>`

- could use lock (`std::atomic_flag`)
- usually really atomic for primitive types
- wide palette of atomic operations

STD – Atomic Flag

Spin lock

```
struct SpinLock {
    SpinLock() { _flag.clear(); }
    ~SpinLock() {
        assert( !_flag.test_and_set() );
    }
    void lock() {
        while( _flag.test_and_set() );
    }
    void unlock() { _flag.clear(); }
private:
    std::atomic_flag _flag;
};
```

STD – Atomic

Pick a Seat

```
S *mySeat = new ...;
for ( std::atomic< S * > &seat : row ) {
    S *expected = nullptr;
    if (seat.compare_exchange_strong(expected, mySeat)) {
        // we can sit down
        break;
    }
    while ( expected->power() < mySeat->power() ) {
        // kick him off
        if ( seat.compare_exchange_strong( expected,
                                           mySeat ) )
            break;
    }
}
```

Lock-Free Programming

- the previous algorithm
- consists of
  - exchanges operations
  - compare and swap operations
  - cycles
- "algorithm is *lock-free* if there is guaranteed system-wide progress"
  - spin lock breaks this condition (deadlock)
- "algorithm is *wait-free* if there is also guaranteed per-thread progress"

Lock-Free Programming

- the previous algorithm
- consists of
  - exchanges operations
  - compare and swap operations
  - cycles
- “algorithm is *lock-free* if there is guaranteed system-wide progress”
  - spin lock breaks this condition (deadlock)
- “algorithm is *wait-free* if there is also guaranteed per-thread progress”
- Wanna know more?
  - Join the Paradise lab.

Task – Lock-Free Queue

- implement a simple lock-free queue
- do not use mutexes, just atomic operations
- use algorithm from this paper: `http://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf`