Atomics

Funkce

Hlavičky `<atomic>` a `<stdatomic.h>` obsahují pár užitečných atomických instrukcí. Mezi ty základní patří:

```
// Uloží do atomické proměnné uložené v obj hodnotu desired
void atomic_store( volatile A* obj , C desired );

// Načte hodnotu obj
C atomic_load( const volatile A* obj );

// Uloží do atomické proměnné uložené v obj hodnotu desired a vrátí její původní
hodnotu
C atomic_exchange( volatile A* obj, C desired );

// Pokud se hodnota uložená v obj rovná expected, nastaví hodnotu na desired.
// Jinak nastaví expected na hodnotu desired
_Bool atomic_compare_exchange_strong( volatile A* obj,
                                      C* expected, C desired );
_Bool atomic_compare_exchange_weak( volatile A* obj,
                                      C* expected, C desired );

// získá hodnotu obj a přičte arg a vrátí novou hodnotu
C atomic_fetch_add( volatile A* obj, M arg );

// získá hodnotu obj a odečte arg a vrátí novou hodnotu
C atomic_fetch_sub( volatile A* obj, M arg );

// získá hodnotu obj a použije operaci or na hodnotu a arg a vrátí novou hodnotu
C atomic_fetch_or( volatile A* obj, M arg );

// získá hodnotu obj a použije operaci xor na hodnotu a arg a vrátí novou hodnotu
C atomic_fetch_xor( volatile A* obj, M arg );
```

Rozdíl, mezi CAS strong a weak je ten, že weak se může chovat jakože `*obj != expected` i když se ve skutečnosti rovnají. Proto se používá například pokud na expected čekáme a nedělá nám problém se zeptat znovu. Weak verze je právě ve smyčce o něco rychlejší než strong.

Memory model

Kompiler pro C a C++ velmi agresivně optimalizuje. Jedna z optimalizací je změna pořadí, ve kterém se přichází do paměti. Například:

```
y = 1
x = 2
y = 3
```

by kompiler mohl optimalizovat jako:

```
y = 3
x = 2
```

V případě paralerního programování ale můžeme počítat s tím, že y je právě 1, nebo je x nejprve 2 a až poté je y rovno 3. V případě atomických operací musíme kompileru říct pořadí přístupů do paměti. K tomu použijeme v každé operaci můžeme použít `explicit` verze atomických operací a definovat tzv. memory order. Rozdělují se na silné a slabé. Ty silné mohou vyžadovat větší režii na

některých platformách a jsou málokdy opravdu třeba. Pokud si ale kódem nejsme jistí, je to možnost. Slabé naopak nevyžadují téměř žádnou režii a jsou většinou zadarmo.

Sequentially consistent
Nejsilnější typ memory orderu. Zajišťuje, že všechny přístupy do paměti před a po této operaci zůstanou před a po.

```
atomic_compare_exchange_explicit(
    &value,
    &expected,
    desired,
    memory_order_seq_cst)
```

Acquire & Release
Dva slabší, navzájem se doplňující, memory ordery. Acquire zajistí, že všechny přístupy do paměti po této operaci, tak zůstanou až po této operaci. Naopak release zajistí, že všechny přístupy do paměti před touto operací zůstanou před touto operací. Používané pro implementaci zámku.

```
atomic_atomic_store_explicit(
    &value,
    1
    memory_order_acquire);

// kritická sekce

atomic_atomic_store_explicit(
    &value,
    0
    memory_order_release);
```

Relaxed
Nejslabší memory order. Nezajišťuje jakékoliv pořadí, pouze to, že se operace vykoná. Používá se pro čítače, které nejsou použitý k synchronizaci. Například počítání hran k uzlu v grafu, kdy při hodnotě 0 chceme uzel odstranit, a je nám celkem jedno kdy se operace provede. Hlavní pro nás je pouze to, že se provede.

```
atomic_atomic_fetch_add_explicit(
    &reference_count,
    1
    memory_order_relaxed);
```

Příklady
Například implementovat semafor:

```
typedef Semaphore _Atomic(int);

void semaphore_wait(Semaphore *pSemaphore) {
  uint32_t expected = 1;
  while(!atomic_compare_exchange_weak_explicit(pSemaphore, &expected, 0,
memory_order_relaxed, memory_order_acquire)) {
    expected = 1;
  }
}

void semaphore_signal(Semaphore *pSemaphore) {
```

```
  atomic_store_explicit(&pSemaphore, 1, memory_order_release);
}
```