Теория и практика многопоточного программирования

Семинар 4

Неганов Алексей

Московский физико-технический институт (национальный исследовательский университет)
Кафедра теоретической и прикладной информатики

Москва 2020

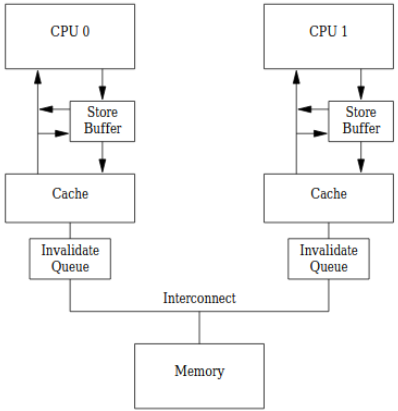
RMW-операции: ABA

LL/SC:

```
word LL(word *ptr) {
   return *ptr; // and set cache line status bit simultaneously
}

bool SC(word *ptr, word newval) {
   if ( /* cache line bit is set */ ) {
      *ptr = newval;
      return true ;
   }
   else
      return false ;
}

bool CAS(word *ptr, word oldval, word newval) {
   if (LL(ptr) == oldval)
      return SC(ptr, newval);
   return false ;
}
```

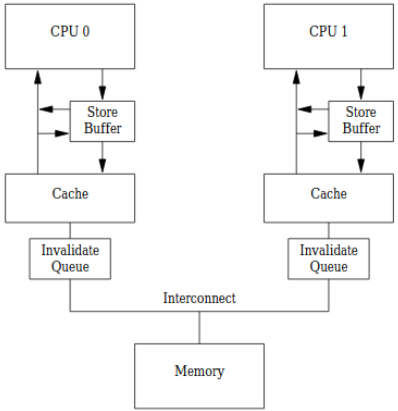
Барьеры памяти

```
int a = 0;
int b = 0;

void foo(void) {
      a = 1;
      b = 1;
}

void bar(void) {
      while(b == 0);
      assert(a == 1);
}
```



Барьеры памяти

```
int a = 0;
int b = 0;

void foo(void) {
      a = 1;
      smp_wmb();
      b = 1;
}

void bar(void) {
      while(b == 0);
      smp_rmb();
      assert(a == 1);
}
```



Модели памяти: relaxed

```
// 'memory_order_relaxed' argument is omitted for brevity
atomic_t x, y;
x.store(0);
y.store(0);

y.store(20);      if (x.load() == 10) {          if (y.load() == 10)
x.store(10);        assert (y.load() == 20)        assert (x.load() == 10);
                    y.store (10);
                  }
```

Load/store операции могут переупорядочиваться, может сработать и тот, и другой `assert`.

```
x.store(1);      y = x.load();
y.store(2);      z = x.load();
                 assert(y <= z);
```

Гарантия – `assert` не может сработать.

Модели памяти: sequentially consistent

```
atomic_t x, y;
x.store(0);
y.store(0);

y.store(20);        if (x.load() == 10) {          if (y.load() == 10)
x.store(10);          assert (y.load() == 20)        assert (x.load() == 10);
                      y.store (10);
                    }
```

Load/store операции могут выполняться лишь в том порядке, что и в коде, ни один `assert` выполниться не может.

```
atomic_t x;
x.load(0);
int y = 0;

y = 1               if (x.load() == 2)
x.store(2);           assert (y == 1)
```

Запись в *y* в коде происходит до записи в *x*, следовательно, `assert` выполниться не может.

Модели памяти: acquire/release

```
load memory, register ;
membar #LoadLoad | #LoadStore ; // acquire barrier

...

membar #LoadStore | #StoreStore ; // release barrier
store regiser, memory
```

Модели памяти: acquire/release

```
#define A memory_order_acquire
#define R memory_order_release

// thread 1                                    // thread 2
y.store(20, R);                                x.store(10, R);

// thread 3                                    // thread 4
assert(y.load(A) == 20 && x.load(A) == 0);     assert(y.load(A) == 0 && x.load(A) == 10);
```

В sequential consistency один из `assert` сработает, в acquire/release — оба могут пройти.

```
// 'memory_order_XXX' argument is omitted for brevity
atomic_t x, y;
x.store(0);
y.store(0);

y.store(20);       if (x.load() == 10) {          if (y.load() == 10)
x.store(10);         assert (y.load() == 20)        assert (x.load() == 10);
                     y.store (10);
                   }
```

В acquire/release `assert` второго потока пройдёт, а третьего — сработает.

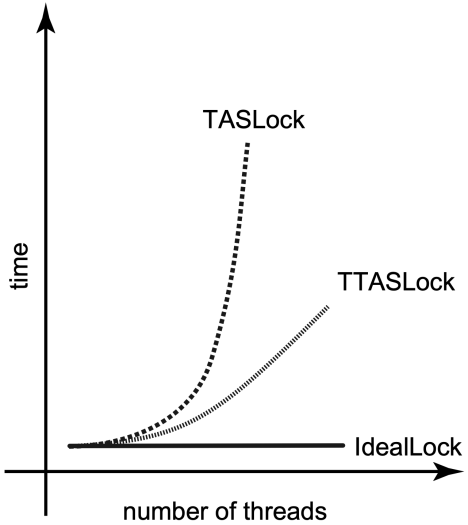
Spinlock

```
class spin_lock_TAS
{
    atomic<unsigned int> m_spin ;
public:
    spin_lock(): m_spin(0) {}
    ~spin_lock() { assert( m_spin.load(memory_order_relaxed) == 0);}

    void lock()
    {
        unsigned int expected;
        do { expected = 0; }
        while ( !m_spin.compare_exchange_weak( expected, 1, memory_order_acquire ));
    }
    void unlock()
    {
        m_spin.store( 0, memory_order_release );
    }
};
```

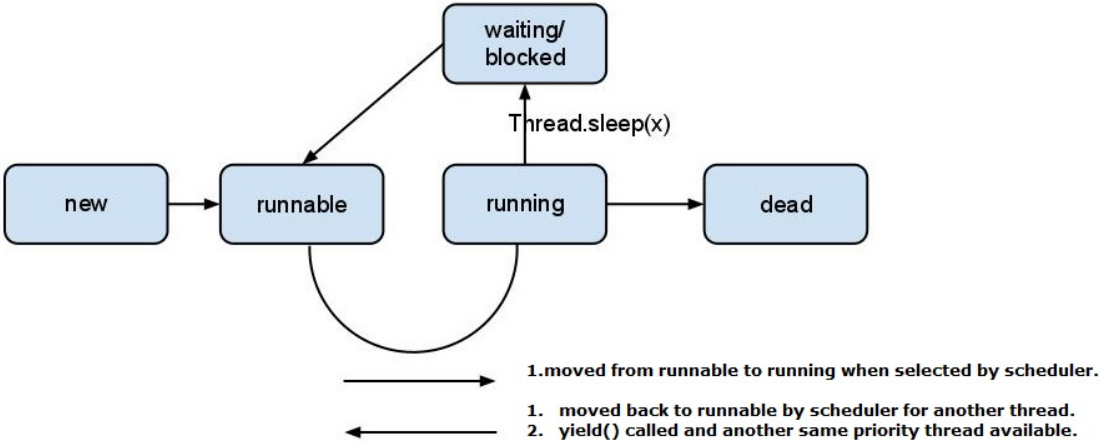
Spinlock

```
class spin_lock_TTAS
{
    atomic<unsigned int> m_spin ;
public:
    spin_lock(): m_spin(0) {}
    ~spin_lock() { assert( m_spin.load(memory_order_relaxed) == 0);}

    void lock()
    {
        unsigned int expected;
        do {
            while (m_spin.load(memory_order_acquire));
            expected = 0;
        }
        while ( !m_spin.compare_exchange_weak( expected, 1, memory_order_acquire ));
    }
    void unlock()
    {
        m_spin.store( 0, memory_order_release );
    }
};
```

Spinlock



Жизненный цикл потока: yield vs sleep



Задачи

1. **(Обязательная)** Напишите свои mutex'ы, использующие yield / exponential backoff. Сравните производительность TAS lock / TTAS lock / lock with yield / backoff lock. Желательно использовать C++11 (и выше), при желании можно GNU C11 и pthreads.

Доклады:

1. Протоколы MESI, MOESI, MESIF.
2. Барьеры памяти в x86 с примерами на ассемблере. Связь с memory order из C11 / C++11.