# Operating Systems

## OOO Execution and hardware assisted synchronization

Me

October 19, 2016

# Алгоритм Петерсона: то, что написали вы

```
1  int turn;
2  int claim[2];
3
4  void lock(int thread)
5  {
6          const int other = (thread + 1) & 1;
7
8          clain[thread] = 1;
9          turn = other;
10
11         while (claim[other] && turn != thread);
12 }
13
14 void unlock(int thread)
15 {
16         claim[thread] = 0;
17 }
```

# Алгоритм Петерсона: то, что уидел компилятор

```
1  .comm claim, 8
2
3  lock:
4          /* eax = thread + 1 */
5          leal    1(%rdi), %eax
6          /* rdx = thread */
7          movslq  %edi, %rdx
8          /* claim[thread] = 1 */
9          movl    $1, claim(,%rdx,4)
10         /* eax = eax & 1, aka, eax = other */
11         andl    $1, %eax
12
13         /* rdx = other */
14         movslq  %eax, %rdx
15
16         /* check if claim[other] == 0 */
17         movl    claim(,%rdx,4), %edx
18         testl   %edx, %edx
19         je      exit
20
21         /* check if other == thread */
22         cmpl    %eax, %edi
23         je      exit
24
25 loop:   jmp     loop
26
27 exit:   ret
```

# Почему???

- Почему компилятор выкинул turn?
  - Компилятор не увидел смысла в сохранении turn и удалил его.
- Почему компилятор сравнивает other с thread?
  - Компилятор выкинул turn, other - это значение, которое должно было лежать в turn.
- Почему компилятор проверяет условия только один раз?
  - Компиялтор не видит, что turn или claim могут измениться, а значит достаточно проверить их один раз и либо выйти либо зациклиться.

# Компилятор виноват?

- Компилятор работает в терминах наблюдаемого поведения:
  - компилятор может делать с вашей программой все что угодно, до тех пор пока наблюдаемое поведение сохраняется;
  - компилятор хочет делать с вашей программой много чего, потому что он должен оптимизировать код.
- Как сделать операции над данными частью наблюдаемого поведения?
  - в языках C и C++ для этого есть ключевое слово *volatile*;
  - компилятор воздерживается от многих оптимизаций кода работающего над volatile объектами.

# Оптимизации компилятора и volatile

- Пометив данные как volatile вы связываете компилятору руки везде, где вы используете volatile данные
  - причем везде - и там где это важно и там где нет;
  - есть ли более точечные средства ограничения фантазии компилятора?
  - обычно они есть, но они зависят от компилятора.
- Барьеры компиялтора:
  - например, в gcc можно использовать такую конструкцию

```
1 __asm__ volatile ("" : : : "memory")
```

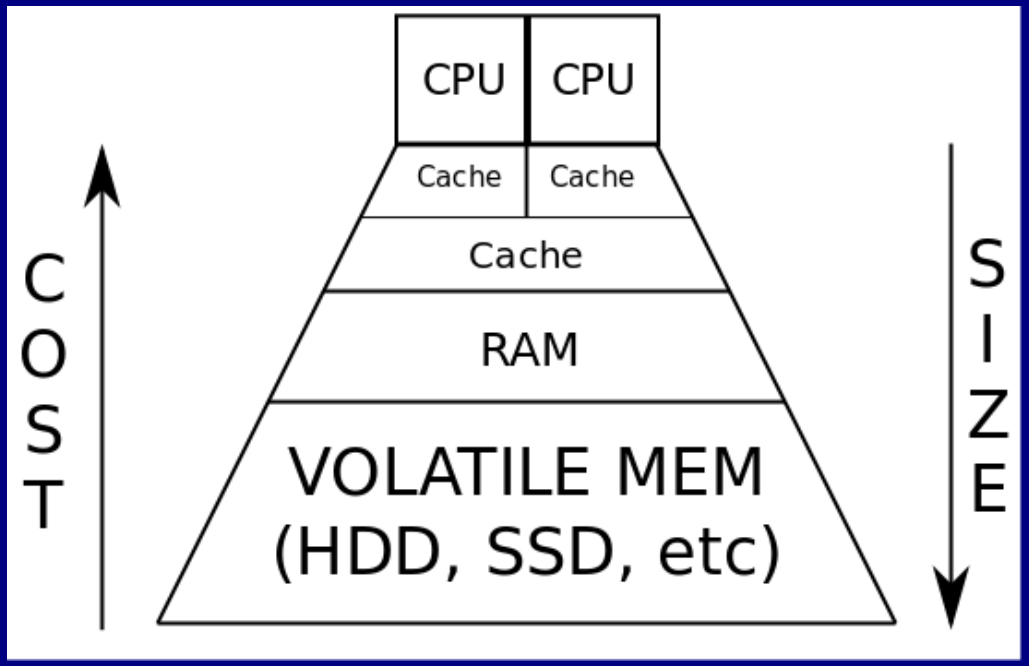
# Алгоритм Петерсона с барьерами компиялтора

```
 1  int turn;
 2  int claim[2];
 3
 4  void lock(int thread)
 5  {
 6          const int other = (thread + 1) & 1;
 7
 8          claim[thread] = 1;
 9          __asm__ volatile ("" : : : "memory"); // order is important here
10          turn = other;
11
12          do {
13                  // reread claim and turn after every iteration
14                  __asm__ volatile ("" : : : "memory");
15          } while (claim[other] && turn != thread);
16  }
17
18  void unlock(int thread)
19  {
20          claim[thread] = 0;
21  }
```

# Замечания про volatile

- volatile не может являться средством синхронизации:
  - volatile не дает никаких гарантий атомарности, т. е. работа с volatile переменными все еще не атомарна;
  - стандарты C и C++ вводят такое понятие как *точка следования*, между двумя точками следования операциям компиялтору разрешено переупорядочивать обращения к volatile объектам;
  - стандарты не определяют строго, что такое "обращение" к volatile объекту;
  - компиялтор - не единственный источник переупорядочивания/оптимизаций, т. е. только компиялторных средств не достаточно.

# Многоуровневая организация памяти




- CPU очень быстрый, а память либо медленная либо дорогая
  - поэтому приходится использовать несколько уровней памяти;
  - чем ближе память к процессору, тем она быстрее и тем ее меньше;

# Принцип локальности

- Временная локальность:
  - когда программа обращается к одним и тем же данным в течение короткого интервала времени;
  - пример: цикл считающий сумму элементов массива - переменная с результатом обновляется на каждый элемент массива.
- Пространственная локальность:
  - когда программа обращается к близким ячейкам памяти в течение короткого интервала времени;
  - пример: цикл считающий сумму элементов массива - мы читаем элементы массива последовательно.
- Зачастую, ваша программа не работает сразу со всеми петабайтами данных одновременно.

# Когерентность кешей

- CPU, зачастую, имеет свой собственный кеш:
  - что если два CPU в своих кешах закеширвоали одну и ту же переменную?
  - ничего плохого, пока копии идентичны;
  - что если один из CPU решил обновить значение у себя в кеше?
  - одна и та же переменная будет иметь разные значения для разных CPU.
- Архитектуры, которые разрешают кешам "расходиться" называются некогерентными
  - не менстрим.

# Протоколы когерентности кешей

- Для обеспечения когерентности кеши могут обмениваться сообщениями
  - CPU соединены между собой полноценной сетью;
  - не такой как TCP/IP/Ethernet, но все еще полноценной.
- Классы сообщений и как нужно на них реагировать - протокол когерентности кешей
  - классический *учебный* пример - протокол MESI.

# Протокол MESI 1/3

- Каждая кеш-линия имеет одно из следующих состояний:
  - M (Modified) - версия данных в кеш линии отличается от того, что хранится в памяти, а кроме того данный кеш является *единственным* владельцем данных;
  - E (Exclusive) - версия данных в кеш линии совпадает с тем, что хранится в памяти, и данный кеш *единственный* владелец данных;
  - S (Shared) - версия данных может находится в других кешах, и совпадает с тем, что хранится в памяти;
  - I (Invalid) - кеш линия не хранит данных.

# Протокол MESI 2/3

- Кеши и память обмениваются друг с другом сообщениями:
  - мы считаем все сообщения широковещательными, т. е. каждый видит все сообщения.
- Состояние кеш линии, обычно, меняется в ответ на получение сообщения.
- Modified может измениться в Exclusive, если процессор решил обновить версию в памяти
  - кеш имеет ограниченный размер и когда-то мы должны из него что-то выбросить;
  - если кеш хранит более новую версию чем память, то перед тем как выбрасить ее из кеша нужно обновить память;
  - любую кеш линии в состоянии S или E можно спокойно выбрасывать из кеша.

# Протокол MESI 3/3

- Типы сообщений:
  - Read - у нас в кеше нет данных, мы хотим прочитать их из памяти или другого кеша;
  - Read Response - ответ на запрос Read, вообще говоря, ответить может кто угодно, но мы будем считать, что отвечает всегда кеш, в котором есть данные или память, если кеша с новой версией данных не нашлось;
  - Invalidate - заставить все кеши сбросить свою версию определенных данных;
  - Invalidate Ack - подтверждение, что другие кеши сбросили свою версию данных, только получив подтверждение мы можем двигаться дальше;
  - Read Invalidate - Read и Invalidate вместе, т. е. не только дайте мне данные, но еще и удалите их у себя.

# Финальные замечания про когерентность кешей

- Фактически, чтобы обновить разделяемые данные необходимо быть их единственным владельцем:
  - перед тем, как мы сможем обновить данные в кеше, кеш линия должна быть в состоянии E или M;
  - запись в общие данные приводит к обмену сообщениями, ожиданию ответов и поэтому стоит дорого;
  - нужно избегать записи в одну и ту же переменную с разных CPU, или ограничивать количество CPU пишущих в одну переменную;
  - другими словами нужно избегать активной конкуренции (contention).

# Store Buffer 1/2

```
#define barrier() \
  __asm__ volatile(
      ↪ "" \
        : \
        : \
        : "memory")

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

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

- Посмотрим на функцию foo:
  - допустим b лежит в кеше CPU, а a нет;
  - CPU пытается присвоить a, но ее нет в кеше, он посылает Invalidate и должен дождаться ответа;
  - но CPU ведь уже знает, что будет лежать в a, ему даже не нужно старое значение, так почему вдруг он должен ждать?

# Store Buffer 2/2

- Если мы не можем писать прямо в кеш, пока не придет Invalidate Ack, то заведем специальный некогерентный буфер для таких записей:
  - такой буфер обычно называют Store Buffer;
  - как только мы получаем Invalidate Ack мы переносим данные из Store Buffer в кеш;
  - сохранив данные в Store Buffer CPU может продолжать работать.
- Не ломает ли Store Buffer что-нибудь?
  - а если сообщения могут переупорядочиваться?

# Теперь CPU виноват? 1/2

```
#define barrier() \
    __asm__ volatile( \
        "" \
        : \
        : \
        : "memory")

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

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

- Рассмотрим следующий порядок событий:
  - CPU0 выполняет foo, а CPU1 выполняет bar;
  - переменная a хранится только в кеше CPU1, а b только в кеше CPU0;
  - CPU0 исполняет $a = 1$, переменной нет в кеше, записываем в Store Buffer и посылаем Invalidate;
  - CPU1 проверяет условие цикла, переменной b нет в кеше, посылаем Read;
  - CPU0 выполняет $b = 1$, b уже в кеше, можно прям там и обновить;
  - CPU0 получает Read и отсылает $b == 1$ в ответ;
  - CPU1 получает значение $b == 1$, и выполняет assert.

# Теперь CPU виноват? 2/2

```
#define barrier() \
  __asm__ volatile(
      ↪ "" \
        : \
        : \
        : "memory")

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

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

- Посмотрим на функцию bar:
  - допустим переменная *a* находится в кеше CPU, а переменная *b* нет;
  - т. е. чтение *a* занимает много меньше времени, чем *b*;
  - если CPU будет ждать пока завершится чтение *b* ему придется остановиться;
  - но у него уже есть *a* и он *не знает* о зависимости между ними, так может обработаем *a* вперед?

# Барьеры памяти

- Откуда вдруг взялось так много проблем?
  - при многопточности, важными становятся зависимости между переменными;
  - например, для функции bar важно, чтобы foo сначала записала a, и только потом b;
  - ни компилятор ни CPU об этих зависимостях ничего не знают.
- Чтобы указать CPU, какие действия нельзя "переставлять" есть специальные инструкции
  - такие инструкции называют барьерами памяти, что конкретно они разершают или запрещают зависит от архитектуры.

# Memory Order

- Правила, по которым CPU может переставлять инструкции зависят от архитектуры, что делать простым смертным?
  - языки программирования предоставляют несколько хорошо известных сравнительно (ну да, конечно...) простых моделей: JMM, sequential consistency, acquire/consume/release;
  - вы можете просто пользоваться обычными mutex-ами реализованными кем-то другим и не иметь проблем с memory order, т. к. lock и unlock делают такими, чтобы они использовали все нужные барьеры памяти/компилятора.

# RMW инструкции и аналоги

- Некоторые архитектуры позволяют прочитать/модифицировать/записать переменную атомарно:
  - атомарные инкремент и декремент;
  - CAS (Compare-And-Swap) - читает значение, сравнивает его с некоторым значением и, если прочитанное значение совпало с аргументом, то записывает новое значение.
- Load Linked/Store Conditional:
  - Load Linked читает значение из памяти;
  - Store Linked записывает значение в память, но только в том случае, если кто-то другой не успел его перезаписать с момента Load Linked.
- При этом налагаются серьезные ограничения
  - в первую очередь на размер операндов.

# Взаимное исключение сиспользованием CAS

```
1  #include <stdatomic.h>
2
3  #define LOCKED   1
4  #define UNLOCKED 0
5
6  struct spinlock {
7    atomic_int locked;
8  };
9
10 void lock(struct spinlock *lock)
11 {
12   while (atomic_exchange_explicit(&lock->locked,
13             LOCKED, memory_order_acquire) == LOCKED);
14 }
15
16 void unlock(struct spinlock *lock)
17 {
18   atomic_store_explicit(&lock->locked, UNLOCKED,
19     memory_order_release);
20 }
```

# Проблемы CAS lock-a

- Честность - никаких гарантий.
- Прозводительность:
  - помните о когерентности кешей, чтобы что-то записать приходиться получить переменную в эксклюзивное пользование;
  - все потоки/CPU будут драться за одну общую переменную пытаясь туда что-то записать;
  - сообщения между кешами будут мешать другим операциям.

# Взаимное исключение сиспользованием CAS, версия 2

```
1  #include <stdatomic.h>
2
3  #define LOCKED   1
4  #define UNLOCKED 0
5
6  struct spinlock {
7    atomic_int locked;
8  };
9
10 void lock(struct spinlock *lock)
11 {
12   do {
13     while (atomic_load_explicit(&lock->locked,
14                 memory_order_relaxed) == LOCKED);
15   } while (atomic_exchange_explicit(&lock->locked,
16               LOCKED, memory_order_acquire) == LOCKED);
17 }
18
19 void unlock(struct spinlock *lock)
20 {
21   atomic_store_explicit(&lock->locked, UNLOCKED,
22     memory_order_release);
23 }
```

# Проблемы 2-ой версии

- Честность - никаких гарантий, опять.
- Прозводительность:
  - теперь потоки сначала читают переменную, несколько CPU могут держать в кеше копии одной переменной, до тех пор, пока один из них не попытается модифицировать перменную;
  - т. е. пока лок занят и мы ждем нам не нужно пересылать сообщения между кешами;
  - но стоит владельцу лока его освободить, нам опятьприходится довольно активно обмениваться сообщениями.

# Ticket Lock

```
1  #include <stdatomic.h>
2
3  struct spinlock {
4    atomic_uint next;
5    atomic_uint current;
6  };
7
8  void lock(struct spinlock *lock)
9  {
10   const unsigned ticket = atomic_fet_add_explicit(&lock->next, 1,
11                                memory_order_relaxed);
12
13   while (atomic_load_explicit(&lock->current,
14             memory_order_acquire) != ticket);
15 }
16
17 void unlock(struct spinlock *lock)
18 {
19   const unsigned current = atomic_load_explicit(&lock->current,
20                                memory_order_relaxed);
21
22   atomic_store_explicit(&lock->current, current + 1,
23     memory_order_release);
24 }
```

# Проблемы Ticket Lock-a

- Честность:
  - все потоки выстраиваются в очередь, согласно порядку инкремента переменной next;
  - с честностью все в порядке!
- Производительность:
  - потоки модифицируют одну общую переменную, но только один раз и после этого ее не трогают;
  - активное ожидание использует только атомарное чтение;
  - при освобождении лока происходит обмен сообщениями.
- А есть ли проблемы?
  - этот алгоритм достаточно хорош, чтобы использоваться в ядре довольно популярной ОС;
  - но мы можем сделать лучше!

# MCS Lock: lock

```
1  #include <stdatomic.h>
2
3  struct node {
4    struct node * _Atomic next;
5    atomic_int wait;
6  };
7
8  struct spinlock {
9    struct node * _Atomic tail;
10 };
11
12 void lock(struct spinlock *lock, struct node *self)
13 {
14   atomic_store_explicit(&self->next, 0, memory_order_relaxed);
15   atomic_store_explicit(&self->wait, 0, memory_order_relaxed);
16
17   struct node *tail = atomic_exchange_explicit(&lock->tail, self,
18                                                memory_order_acq_rel);
19
20   if (!tail)
21     return;
22
23   atomic_store_explicit(&tail->next, self, memory_order_relaxed);
24   while (!atomic_load_explicit(&self->wait, memory_order_acquire));
25 }
```

# MCS Lock: unlock

```
1  void unlock(struct spinlock *lock, struct node *self)
2  {
3    struct node *tail = self, *next;
4
5    if (atomic_compare_exchange_strong_explicit(&lock->tail, &tail, 0,
6          memory_order_release, memory_order_relaxed))
7      return;
8
9    while (!(next = atomic_load_explicit(&self->next,
10                    memory_order_relaxed)));
11   atomic_store_explicit(&next->wait, 0, memory_order_release);
12 }
```

# MCS Lock: зачем так сложно?

- Очевидно, MCS Lock гарантирует честность:
  - все потоки выстравиваются в очередь;
  - причем в прямом смысле: у нас есть связный список ожидающих.
- Не более двух потоков используют одну переменную *wait*:
  - при unlock-е Invalidate достаточно отослать только одному CPU (если его можно отослать только одному CPU);
  - отреагировать на Invalidate должен только один CPU.
- Статья (с кучей другой информации о локах):
  - Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors.

# Локи и прерывания

- Рассмотрим следующий сценарий:
  - поток захватил лок, и работает с разделяемой структурой памяти;
  - поток прерывается обработчиком прерываний, и так случилось, что обработчик так же хочет захватить тот же самый лок;
  - что произойдет?
    - если нас не прервет какое-то другое прерывание, то это deadlock!
- Мораль: выключайте прерывания, перед захватом локов
  - если лок очень загружен, то захват может занять какое-то время;
  - все это время прерывания на CPU будут выключена;
  - мораль: избегайте контеншена.

# Блокировка потоков

- Рассмотренные локи в коде назывались spinlock:
  - от слова spin, что в данном случае значит активное ожидание;
  - т. е. мы "крутимся" в цикле и не делаем ничего полезного;
  - это не плохо, если ждать нам не долго, но что если все не так?
- Мы можем сообщить планировщику, что поток ждет
  - т. е. мы можем снять поток с CPU и пометить его как заблокированный;
  - планировщик не будет давать ему CPU, пока поток не разблокируют;
  - такую реализацию часто называют mutex в различных библиотеках.

# Q&A