

Operating Systems

Threads Synchronization

Me

March 4, 2016

- 1 Конкурентное исполнение. Состояние Гонки.
- 2 Взаимное исключение. Алгоритм Петерсона.
- 3 Честное взаимное исключение. Алгоритм пекарни.
- 4 Out-of-order execution. Когерентность кешей и модели памяти.
- 5 Атомарность (CAS и LL/SC). Test-And-Set Lock. Queued Locks.
- 6 Атомарный снимок (seqlock).
- 7 Неблокирующая синхронизация.
- 8 Проблема ABA и безопасное освобождение памяти.
- 9 Differential Reference Counting. Hazzard Pointers. RCU.
- 10 Per-CPU и Thread Local данные.

Конкурентное исполнение



Figure : Concurrent Execution

Конкурентное исполнение - исполняющиеся участки кода накладываются друг на друга произвольным образом

- вы не должны делать никаких предположений о порядке наложения;
- произвольный порядок ведет к произвольному доступу к общим ресурсам;

Конкурентное исполнение

Источники конкурентности

- много агентов исполняющих код:
 - Hyper Threading, SMP, NUMA (shared memory);
 - cluster nodes (shared storage);

Конкурентное исполнение

Источники конкурентности

- много агентов исполняющих код:
 - Hyper Threading, SMP, NUMA (shared memory);
 - cluster nodes (shared storage);
- прерывания - прерывают один код и запускают исполнение другого;

Конкурентное исполнение

Источники конкурентности

- много агентов исполняющих код:
 - Hyper Threading, SMP, NUMA (shared memory);
 - cluster nodes (shared storage);
- прерывания - прерывают один код и запускают исполнение другого;
- сигналы - userspace аналог прерываний.

Конкурентное исполнение

Состояние гонки

```
1 int cnt;  
2  
3 void foo(void)  
4 {  
5     ++cnt;  
6 }
```

```
1 extern int cnt;  
2  
3 void bar(void)  
4 {  
5     ++cnt;  
6 }
```

Конкурентное исполнение

Состояние гонки

```
1  .global cnt
2  cnt:
3  .int 0
4
5  foo:
6      mov cnt, %rax
7      inc %rax
8      mov %rax, cnt
9      ret
```

```
1  .extern cnt
2
3  bar:
4      mov cnt, %rax
5      inc %rax
6      mov %rax, cnt
7      ret
```


Взаимное исключение

Взаимное исключение (Mutual Exclusion) - позволяет оградить критическую секцию, чтобы предотвратить конкуренции

- lock и unlock - ограничивают критическую секции в начале и конце соответственно;
- только один поток исполнения может находиться в критической секции
 - lock не вернет управление, до тех пор, пока поток в критической секции не сделает unlock;
- если в критической секции не находится поток, то из нескольких конкурирующих lock-ов, как минимум один будет успешным;

Взаимное исключение

Мы можем считать, что

- поток выйдет из критической секции за конечное время
 - поток не зависнет и не упадет внутри критической секции;

Взаимное исключение

Мы можем считать, что

- поток выйдет из критической секции за конечное время
 - поток не зависнет и не упадет внутри критической секции;

Мы не можем делать предположений о

- скорости работы потока
 - время нахождения в критической секции конечно, но ограничение сверху нам не известно;
- взаимной скорости работы потоков
 - мы не можем считать, что один поток быстрее/медленнее другого или что их скорости равны

Реализация взаимного исключения

Глобальный флаг

```
1 extern int claim1;  
2 int claim0;  
3  
4 void lock0()  
5 {  
6     claim0 = true;  
7     while (claim1);  
8 }  
9  
10 void unlock0(void)  
11 {  
12     claim0 = false;  
13 }
```

```
1 extern int claim0;  
2 int claim1;  
3  
4 void lock1(void)  
5 {  
6     claim1 = true;  
7     while (claim0);  
8 }  
9  
10 void unlock1(void)  
11 {  
12     claim1 = false;  
13 }
```

Реализация взаимного исключения

Глобальный флаг

Следующее расписание приводит к deadlock-у:

- 1 Thread 0, line 6;
- 2 Thread 1, line 6;
- 3 Thread 0, line 7 (Thread 0 завис на этой строке);
- 4 Thread 1, line 7 (Thread 1 завис на этой строке);

Реализация взаимного исключения

Глобальный порядок

```
1  int turn;
2
3  void lock0(void)
4  {
5      while (turn != 0);
6  }
7
8  void unlock0(void)
9  {
10     turn = 1;
11 }
```

```
1  extern int turn;
2
3  void lock1(void)
4  {
5      while (turn != 1);
6  }
7
8  void unlock0(void)
9  {
10     turn = 0;
11 }
```

Реализация взаимного исключения

Глобальный порядок

Расписание приводящее к проблемам:

- 1 Thread 1, line 5 (Thread 1 завис на этой строке);
- 2 Thread 0 - умер (решил не заходить в критическую секцию);

Реализация взаимного исключения

Глобальный порядок

Расписание приводящее к проблемам:

- 1 Thread 1, line 5 (Thread 1 завис на этой строке);
- 2 Thread 0 - умер (решил не заходить в критическую секцию);

Да, оно довольно короткое...

Реализация взаимного исключения

Соберем все в кучу

```
1 extern int claim1;
2 int claim0;
3 int turn;
4
5 void lock0(void)
6 {
7     claim0 = true;
8     turn = 1;
9
10    while (claim1 && turn == 1);
11 }
12
13 void unlock0(void)
14 {
15     claim0 = false;
16 }
```

```
1 extern int claim0;
2 extern int turn;
3 int claim1;
4
5 void lock1(void)
6 {
7     claim1 = true;
8     turn = 0;
9
10    while (claim0 && turn == 0);
11 }
12
13 void unlock1(void)
14 {
15     claim1 = false;
16 }
```

Реализация взаимного исключения

Алгоритм Петтерсона

Доказательство взаимного исключения:

- пусть сразу два потока находятся в критической секции:
 - turn принимает одно из двух значений: 0 или 1; для определенности пусть это будет 0, т. е. последним в turn записывал поток 1;
 - claim0 и claim1 оба равны true;
- к моменту проверки условия цикла потоком 1 имеем:
 - turn равен 0;
 - claim0 равен true;
 - но в этом случае поток 1 должен зависнуть в цикле до изменения turn или claim0 - противоречие;

Реализация взаимного исключения

Алгоритм Петтерсона

Доказательство наличия прогресса: пусть поток 0 пытается войти в свободную критическую секцию:

- поток 0 в 10 строке видит $\text{claim1} == \text{true}$:
 - поток 0 видит $\text{turn} == 0$, поток 0 входит в критическую секцию;
 - поток 0 видит $\text{turn} == 1$, возможны два случая:
 - поток 1 выполнил строку 8, поток 0 перезаписал turn - поток 1 входит в критическую секцию;
 - поток 1 собирается выполнить строку 8 - поток 0 войдет в критическую секцию, после того как поток 1 выполнит строку 8;

Реализация взаимного исключения

Алгоритм Петтерсона

Доказательство наличия прогресса: пусть поток 0 пытается войти в свободную критическую секцию:

- поток 0 в 10 строке видит $\text{claim1} == \text{true}$:
 - поток 0 видит $\text{turn} == 0$, поток 0 входит в критическую секцию;
 - поток 0 видит $\text{turn} == 1$, возможны два случая:
 - поток 1 выполнил строку 8, поток 0 перезаписал turn - поток 1 входит в критическую секцию;
 - поток 1 собирается выполнить строку 8 - поток 0 войдет в критическую секцию, после того как поток 1 выполнит строку 8;
- поток 0 видит $\text{claim1} == \text{false}$ - поток 0 входит в критическую секцию;

Реализация взаимного исключения

Алгоритм Петтерсона для N потоков

```
1  int flag[N];
2  int turn[N - 1];
3
4  void lock(int i)
5  {
6      for (int count = 0; count < N - 1; ++count) {
7          flag[i] = count + 1;
8          turn[count] = i;
9
10         int found = true;
11         while (turn[count] == i && found) {
12             found = false;
13             for (int k = 0; !found && k != N; ++k) {
14                 if (k == i) continue;
15                 found = flag[k] > count;
16             }
17         }
18     }
19 }
20
21 void unlock(int i)
22 {
23     flag[i] = 0;
24 }
```

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Рассмотрим пример на 3 потоках. Начальное состояние:

- $\text{flag}[3] = \{0, 0, 0\};$
- $\text{turn}[2] = \{0, 0\};$

Поток 0 пытается войти в критическую секцию ($\text{count} = 0$):

- $\text{flag}[3] = \{1, 0, 0\};$
- $\text{turn}[2] = \{0, 0\};$

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Поток 1 пытается войти в критическую секцию ($\text{count} = 0$):

- $\text{flag}[3] = \{1, 1, 0\};$
- $\text{turn}[2] = \{1, 0\};$

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Поток 2 пытается войти в критическую секцию ($\text{count} = 0$):

- $\text{flag}[3] = \{1, 1, 1\};$
- $\text{turn}[2] = \{2, 0\};$

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Поток 1 пытается войти в критическую секцию ($\text{count} = 1$):

- $\text{flag}[3] = \{1, 2, 1\};$
- $\text{turn}[2] = \{2, 1\};$

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Поток 1 вошел в критическую секцию... И вышел из критической секции:

- $\text{flag}[3] = \{1, 0, 1\};$
- $\text{turn}[2] = \{2, 1\};$

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Поток 1 пытается войти в критическую секцию ($\text{count} = 0$):

- $\text{flag}[3] = \{1, 1, 1\};$
- $\text{turn}[2] = \{1, 1\};$

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Поток 2 пытается войти в критическую секцию ($\text{count} = 1$):

- $\text{flag}[3] = \{1, 1, 2\};$
- $\text{turn}[2] = \{1, 2\};$

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Поток 2 вошел в критическую секцию... И вышел из критической секции:

- $\text{flag}[3] = \{1, 1, 0\};$
- $\text{turn}[2] = \{1, 2\};$

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Поток 2 пытается войти в критическую секцию ($\text{count} = 0$):

- $\text{flag}[3] = \{1, 1, 1\};$
- $\text{turn}[2] = \{2, 2\};$

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Поток 1 пытается войти в критическую секцию ($\text{count} = 1$):

- $\text{flag}[3] = \{1, 2, 1\};$
- $\text{turn}[2] = \{2, 1\};$

Мы уже были в этом состоянии! А поток 0 так и не получил управление!

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Как определить честность? Разделим lock на две части:

- вход (D) - всегда завершается за известное конечное количество шагов;
- ожидание (W) - может потребовать неограниченное количество шагов;

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Как определить честность? Разделим lock на две части:

- вход (D) - всегда завершается за известное конечное количество шагов;
- ожидание (W) - может потребовать неограниченное количество шагов;

Свойство r -ограниченного ожидания для двух потоков (0 и 1):

- если D_0^k (k -ый вход потока 0) предшествует D_1^j (j -ому входу потока 1);
- тогда k -ая критическая секция потока 0, предшествует $j + r$ -ой критической секции потока 1;

Реализация взаимного исключения

Честность Алгоритма Петтерсона

Как определить честность? Разделим lock на две части:

- вход (D) - всегда завершается за известное конечное количество шагов;
- ожидание (W) - может потребовать неограниченное количество шагов;

Свойство r -ограниченного ожидания для двух потоков (0 и 1):

- если D_0^k (k -ый вход потока 0) предшествует D_1^j (j -ому входу потока 1);
- тогда k -ая критическая секция потока 0, предшествует $j + r$ -ой критической секции потока 1;

Алгоритм Петтерсона не обладает свойством r -ограниченного ожидания ни для какого r .

Реализация взаимного исключения

Алгоритм Пекарни (Л. Лэмпорт)

Каждый поток при попытке входа выбирает себе число:

- число определяет место в очереди;
- новое число выбирается так, чтобы оно было больше всех чисел в очереди;

Реализация взаимного исключения

Алгоритм Пекарни (Л. Лэмпорт)

Каждый поток при попытке входа выбирает себе число:

- число определяет место в очереди;
- новое число выбирается так, чтобы оно было больше всех чисел в очереди;

Как выбирать это число?

- посмотреть на числа всех потоков и прибавить 1 к наибольшему;
- что если два потока выбирают число одновременно?

Реализация взаимного исключения

Алгоритм Пекарни (Л. Лэмпорт)

Каждый поток при попытке входа выбирает себе число:

- число определяет место в очереди;
- новое число выбирается так, чтобы оно было больше всех чисел в очереди;

Как выбирать это число?

- посмотреть на числа всех потоков и прибавить 1 к наибольшему;
- что если два потока выбирают число одновременно?

Как использовать выбранное число?

- если число наименьшее среди всех потоков выбравших число, то входим в критическую секцию;

Реализация взаимного исключения

Алгоритм Пекарни (Л. Лэмпорт)

```
1  int flag[N];
2  int number[N];
3
4  int max(void)
5  {
6      int rc = 0;
7
8      for (int i = 0; i != N; ++i) {
9          const int n = number[i];
10
11          if (n > rc)
12              rc = n;
13      }
14
15      return rc;
16  }
17
18  int less(int id0, int n0,
19           int id1, int n1)
20  {
21      if (n0 < n1)
22          return true;
23      if (n0 == n1 && id0 < id1)
24          return true;
25      return false;
26  }
```

```
1  void lock(int i)
2  {
3      flag[i] = true;
4      number[i] = max() + 1;
5      flag[i] = false;
6
7      for (int j = 0; j != N; ++j) {
8          if (j == i)
9              continue;
10
11          while (flag[j]);
12          while (number[j] &&
13                 less(j, number[j],
14                     i, number[i]));
15      }
16  }
17
18  void unlock(int i)
19  {
20      number[i] = 0;
21  }
```

Реализация взаимного исключения

Честность алгоритм Пекарни

- вход алгоритма пекарни (D) состоит из:
 - выбора нового числа для потока;
- если D_0^k предшествует D_1^j , то число выбранное потоком 0 на входе k , будет меньше числа, выбранного потоком 1 на входе j ;

Реализация взаимного исключения

Честность алгоритм Пекарни

- вход алгоритма пекарни (D) состоит из:
 - выбора нового числа для потока;
- если D_0^k предшествует D_1^j , то число выбранное потоком 0 на входе k , будет меньше числа, выбранного потоком 1 на входе j ;
- т. е. поток 0 войдет в k -ую критическую секцию раньше, чем поток 1 войдет в j -ую - 0-ограниченное ожидание.

Out-of-order execution

К сожалению, описанные подходы, как есть, не будут работать...

Out-of-order execution

К сожалению, описанные подходы, как есть, не будут работать...

- компилятору *разрешено* переставлять инструкции:
 - компилятор может делать с кодом все, что угодно, пока наблюдаемое поведение остается неизменным;
 - кеширование, удаление "мертвого" кода, спекулятивные записи и чтения и многое другое

Out-of-order execution

К сожалению, описанные подходы, как есть, не будут работать...

- компилятору *разрешено* переставлять инструкции:
 - компилятор может делать с кодом все, что угодно, пока наблюдаемое поведение остается неизменным;
 - кеширование, удаление "мертвого" кода, спекулятивные записи и чтения и многое другое
- процессоры могут использовать оптимизации изменяющие порядок работы с памятью:
 - store buffer - сохранение данных во временный буфер вместо кеша;
 - invalidate queue - отложенный сброс линии кеша;

Оптимизации компилятора

Компилятор подбирает оптимальный набор инструкций реализующий заданное наблюдаемое поведение (осторожно С и С++):

- обращения к volatile данным (чтение и запись);
- операции ввода/вывода (printf, scanf и тд).

Оптимизации компилятора

Компилятор подбирает оптимальный набор инструкций реализующий заданное наблюдаемое поведение (осторожно C и C++):

- обращения к `volatile` данным (чтение и запись);
- операции ввода/вывода (`printf`, `scanf` и тд).

Если компилятору не сообщить, то он не знает:

- что переменная может модифицироваться в другом потоке;
- что переменную может читать другой поток;
- что порядок обращений к переменным важен;

Барьеры компилятора

- Чтобы сообщить компилятору о "побочных" эффектах работы с памятью нужно сделать эту память частью наблюдаемого поведения - использовать ключевое слово `volatile`;
 - компилятору запрещено переставлять обращения к `volatile` данным, *если они разделены точкой следования*;
 - компилятор может переставлять доступ к `volatile` данным с доступом к не `volatile` данным;

Барьеры компилятора

```
1 struct some_struct {
2     int a, b, c;
3 };
4
5 struct some_struct * volatile
6     ↪ public;
7
8 void foo(void)
9 {
10     struct some_struct *ptr =
11         ↪ alloc_some_struct();
12
13     ptr->a = 1;
14     ptr->b = 2;
15     ptr->c = 3;
16     // need something to prevent
17     // reordering
18     public = ptr;
19 }
```

```
1 void bar(void)
2 {
3     while (!public);
4     // and here too
5     assert(public->a == 1);
6     assert(public->b == 2);
7     assert(public->c == 3);
8 }
```

Барьеры компилятора

Итого: volatile мало чем помогает, что делать?

Смотреть в документацию компилятора! Например, gcc предлагает следующее решение:

```
1 #define barrier() asm volatile ("" : : : "memory")
```


Барьеры компилятора

```
1 struct some_struct {
2     int a, b, c;
3 };
4
5 #define barrier() asm volatile
6     ↪ ("": : : "memory")
7 struct some_struct *public;
8
9 void foo(void)
10 {
11     struct some_struct *ptr =
12         ↪ alloc_some_struct();
13
14     ptr->a = 1;
15     ptr->b = 2;
16     ptr->c = 3;
17     barrier();
18     public = ptr;
19 }
```

```
1 void bar(void)
2 {
3     while (!public);
4     barrier();
5     assert(public->a == 1);
6     assert(public->b == 2);
7     assert(public->c == 3);
8 }
```

Когерентность процессорных кешей

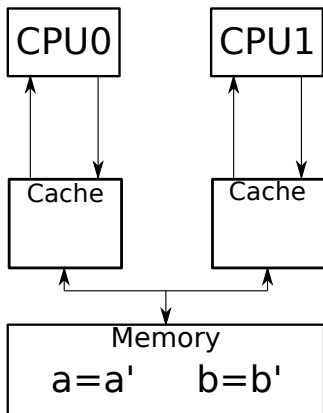


Figure : Cache Incoherency

Когерентность процессорных кешей

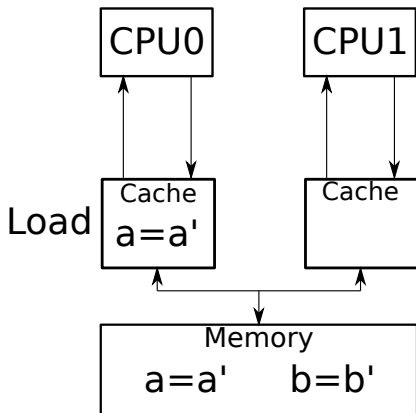


Figure : Cache Incoherency

Когерентность процессорных кешей

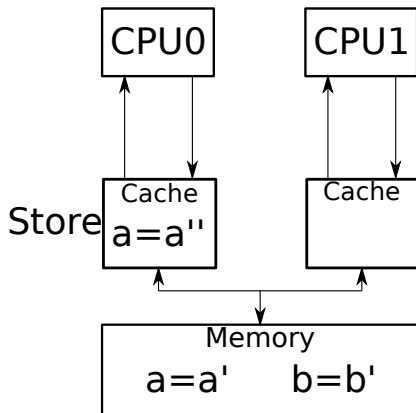


Figure : Cache Incoherency

Когерентность процессорных кешей

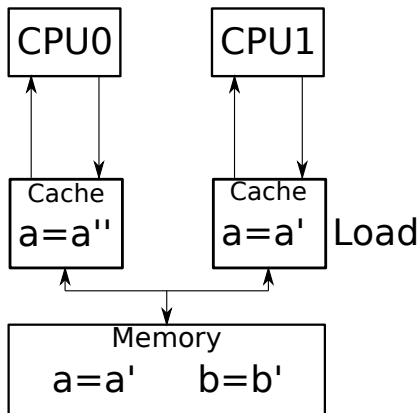


Figure : Cache Incoherency

Когерентность процессорных кешей

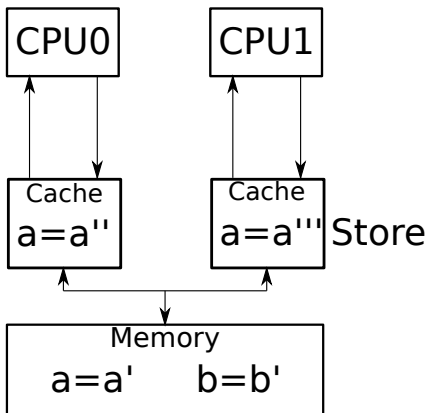


Figure : Cache Incoherency

Когерентность процессорных кешей

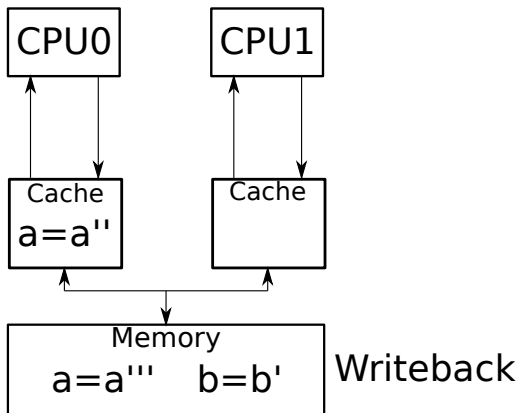


Figure : Cache Incoherency

Когерентность процессорных кешей

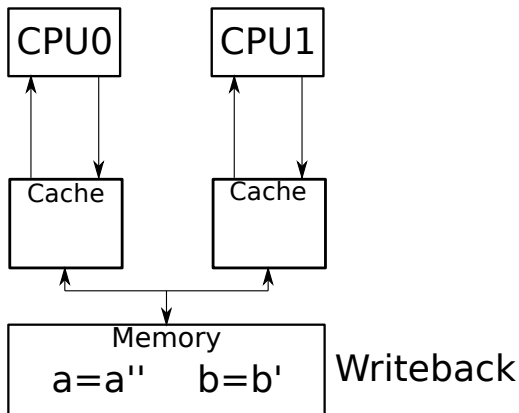


Figure : Cache Incoherency

Когерентность процессорных кешей

Кеши должны находиться в согласованном состоянии (быть когерентными):

- процессоры могут обмениваться сообщениями:
 - можем считать, что сообщения передаются надежно;
 - не можем полагаться на порядок доставки и обработки сообщений;
- процессоры используют специальный протокол обеспечения когерентности:
 - наверно, самый широко известный протокол - MESI (есть сомнения, что он используется без модификаций);

MESI (Modified, Exclusive, Shared, Invalid) предполагает, что каждая линия кеша находится в одном из четырех состояний:

- Modified - кеш линия находится только в кеше данного процессора и она была записана (может отличаться от версии в памяти);
- Exclusive - кеш линия находится только в кеше данного процессора и она совпадает с копией в памяти;
- Shared - кеш линия находится в кеше данного процессора и возможно в кешах других процессоров, содержимое совпадает с памятью;
- Invalid - кеш линия не используется;

Перед тем как модифицировать данные процессор должен получить данные в эксклюзивное пользование:

- если несколько процессоров держат в кеше данные (Shared), то мы просим их сбросить данные;
- если другой процессор держит данные в кеше (Modified или Exclusive), то мы просим его передать нам данные
 - контроллер памяти может увидеть передачу и обновить данные в памяти;
 - или процессор может явно сбросить данные в память;
- если никто не держит данные в кеше, то мы получаем их от контроллера памяти;

Store Buffer

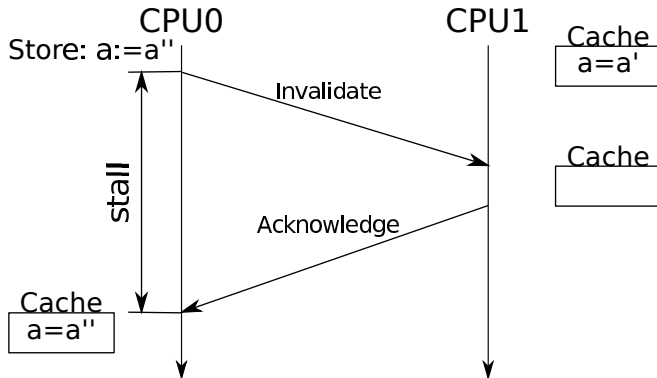


Figure : Cache Store Stall

"Первая" запись данных приводит к ненужным остановкам конвейера:

- другие процессоры должны сбросить данные из кеша;
- процессор должен дожидаться от них подтверждения;

"Первая" запись данных приводит к ненужным остановкам конвейера:

- другие процессоры должны сбросить данные из кеша;
- процессор должен дожидаться от них подтверждения;
- но нам даже не нужно знать старое значение, если мы все равно его перезаписываем!

"Первая" запись данных приводит к ненужным остановкам конвейера:

- другие процессоры должны сбросить данные из кеша;
- процессор должен дожидаться от них подтверждения;
- но нам даже не нужно знать старое значение, если мы все равно его перезаписываем!
- заведем маленький кеш без поддержки когерентности (Store Buffer):
 - первоначально запишем данные в него;
 - когда придет подтверждение - сбросим данные в кеш;

Store Buffer

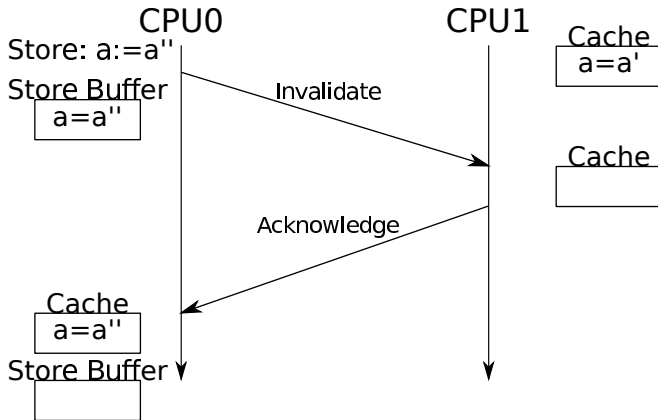


Figure : Store Buffer

Memory Order Violation

```
1 void foo(void)
2 {
3     a = 1;
4     barrier();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10    while (b == 0)
11        continue;
12    barrier();
13    assert(a == 1);
14 }
```

- a равна 0 и находится в кеше CPU1;
- b равна 0 и находится в кеше CPU0;
- CPU0 исполняет `foo` и CPU1 исполняет `bar`;

Memory Order Violation

```
1 void foo(void)
2 {
3     a = 1;
4     barrier();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10    while (b == 0)
11        continue;
12    barrier();
13    assert(a == 1);
14 }
```

- CPU0 исполняет строку 3;
- переменная *a* не в кеше CPU0 - посылаем Invalidate;
- сохраняем новое значение для *a* в Store Buffer;

Memory Order Violation

```
1 void foo(void)
2 {
3     a = 1;
4     barrier();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10     while (b == 0)
11         continue;
12     barrier();
13     assert(a == 1);
14 }
```

- CPU1 исполняет строку 10;
- переменная *b* не в кеше CPU1
- запрашиваем ее для чтения;

Memory Order Violation

```
1 void foo(void)
2 {
3     a = 1;
4     barrier();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10    while (b == 0)
11        continue;
12    barrier();
13    assert(a == 1);
14 }
```

- CPU0 исполняет строку 5;
- переменная *b* лежит в кеше CPU0 - можно ее прямо там и обновить (кеш линия Modified или Exclusive);

Memory Order Violation

```
1 void foo(void)
2 {
3     a = 1;
4     barrier();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10    while (b == 0)
11        continue;
12    barrier();
13    assert(a == 1);
14 }
```

- CPU0 получает запрос на чтение b от CPU1;
- CPU0 отправляет последнее значение $b = 1$;
- CPU0 помечает кеш линию с переменной b как Shared;

Memory Order Violation

```
1 void foo(void)
2 {
3     a = 1;
4     barrier();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10     while (b == 0)
11         continue;
12     barrier();
13     assert(a == 1);
14 }
```

- CPU1 получает ответ от CPU0 со значением b ;
- CPU1 помещает значение b в кеш (кеш линия Shared);
- CPU1 может закончить выполнение строки 10 - условие ложно;

Memory Order Violation

```
1 void foo(void)
2 {
3     a = 1;
4     barrier();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10    while (b == 0)
11        continue;
12    barrier();
13    assert(a == 1);
14 }
```

- CPU1 исполняет строку 12;
- CPU1 держит в кеше старое значение $a = 0$;

Memory Order Violation

```
1 void foo(void)
2 {
3     a = 1;
4     barrier();
5     b = 1;
6 }
7
8 void bar(void)
9 {
10    while (b == 0)
11        continue;
12    barrier();
13    assert(a == 1);
14 }
```

- CPU1 получает Invalidate - но уже поздно;
- CPU1 сбрасывает линию и посылает Acknowledge CPU0;

Store Barrier

- Процессор ничего не знает о зависимостях между переменными
 - он ничего не знал, о том, что сохранение *a* должно предшествовать сохранению *b*;
- чтобы указать процессору на зависимость используется специальная инструкция-барьер
 - в x86 есть инструкция *sfence*, которая гарантирует, что все записи начатые перед барьером "завершаться";
 - другими словами *sfence* ждет, пока Store Buffer опустеет;
 - *sfence* сериализует только store операции;

Invalidate Queue

- Store Buffer имеет ограниченный размер и может переполниться
 - хочется получать Acknowledge на Invalidate побыстрее;
 - сброс данных из кеша может занять время (если кеш занят или если много Invalidate сообщений пришло за раз);
- процессор может отложить инвалидацию кеша и послать Acknowledge почти сразу
 - при этом, конечно, он должен воздержаться от общения с другими CPU об "инвалидированной" кеш линии;
 - Invalidate при этом ставится в очередь (CPU обращается к этой очереди, только если собирается послать сообщение кому-то);

Memory Order Violation

```
1 #define wmb() asm volatile ("
    ↪ sfence" : : : "memory
    ↪ ")
2
3 void foo(void)
4 {
5     a = 1;
6     wmb();
7     b = 1;
8 }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- $a = 0$ и она находится в кеше обоих процессоров (Shared);
- $b = 0$ и она находится в кеше CPU0 (Exclusive или Modified);
- CPU0 исполняет foo, а CPU1 исполняет bar;

Memory Order Violation

```
1 #define wmb() asm volatile ("
    ↪ sfence" : : : "memory
    ↪ ")
2
3 void foo(void)
4 {
5     a = 1;
6     wmb();
7     b = 1;
8 }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- CPU0 исполняет строку 5;
- кеш линия помечена как Shared - нужно послать Invalidate;

Memory Order Violation

```
1  #define wmb() asm volatile ("
    ↪ sfence" : : : "memory
    ↪ ")
2
3  void foo(void)
4  {
5      a = 1;
6      wmb();
7      b = 1;
8  }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- CPU1 исполняет строку 12;
- *b* не в кеше CPU1 -
запрашиваем значение *b*;

Memory Order Violation

```
1 #define wmb() asm volatile ("
    ↪ sfence" : : : "memory
    ↪ ")
2
3 void foo(void)
4 {
5     a = 1;
6     wmb();
7     b = 1;
8 }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- CPU1 получает Invalidate от CPU0;
- CPU1 сохраняет запись в Invalidate Queue, но *не сбрасывает* кеш линию;
- CPU1 отправляет Acknowledge CPU0;

Memory Order Violation

```
1  #define wmb() asm volatile ("
    ↪ sfence" : : : "memory
    ↪ ")
2
3  void foo(void)
4  {
5      a = 1;
6      wmb();
7      b = 1;
8  }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- CPU0 получает Acknowledge от CPU1;
- CPU0 может переместить значение *b* из Store Buffer в кеш и может завершить выполнение барьера;

Memory Order Violation

```
1  #define wmb() asm volatile ("
    ↪ sfence" : : : "memory
    ↪ ")
2
3  void foo(void)
4  {
5      a = 1;
6      wmb();
7      b = 1;
8  }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- CPU0 выполняет строку 7;
- *b* уже в кеше CPU0 и CPU0 владеет этими данными - можно обновить прямо в кеше;

Memory Order Violation

```
1 #define wmb() asm volatile ("
    ↪ sfence" : : : "memory
    ↪ ")
2
3 void foo(void)
4 {
5     a = 1;
6     wmb();
7     b = 1;
8 }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- CPU0 получает запрос на чтение *b* из CPU1;
- CPU0 отправляет обновленное значение *b*;

Memory Order Violation

```
1 #define wmb() asm volatile ("
    ↪ sfence" : : : "memory
    ↪ ")
2
3 void foo(void)
4 {
5     a = 1;
6     wmb();
7     b = 1;
8 }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- CPU1 получает ответ на запрос на чтение *b* от CPU0;
- CPU1 сохраняет полученное *b* в кеш и может завершить проверку условия - условие ложно;

Memory Order Violation

```
1 #define wmb() asm volatile ("
   ↪ sfence" : : : "memory
   ↪ ")
2
3 void foo(void)
4 {
5     a = 1;
6     wmb();
7     b = 1;
8 }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- CPU1 выполняет строку 15;
- CPU1 старое значение $a = 0$ все еще в кеше (мы не поместили ее как Invalid, а сразу отправили подтверждение);

Memory Order Violation

```
1 #define wmb() asm volatile ("
    ↪ sfence" : : : "memory"
    ↪ ")
2
3 void foo(void)
4 {
5     a = 1;
6     wmb();
7     b = 1;
8 }
9
10 void bar(void)
11 {
12     while (b == 0)
13         continue;
14     barrier();
15     assert(a == 1);
16 }
```

- CPU1 обрабатывает отложенный Invalidate, но слишком поздно;

Invalidate Queue

- Процессор ничего не знает о зависимостях между переменными
 - он ничего не знал, о том, что чтение b должно строго предшествовать чтению a и "прочитал a " заранее;
- чтобы указать процессору на зависимость используется специальная инструкция-барьер
 - в x86 есть инструкция *lfence*, которая запрещает переставлять операции чтения;
 - другими словами *lfence* ждет, пока Invalidate Queue опустеет;

Важное замечание касательно примеров:

- в примерах выше sfence не нужен:
 - архитектура x86 гарантирует, что store операции одного процессора не могут быть "переставлены";
- в примерах выше lfence не нужен:
 - архитектура x86 гарантирует, что load операции одного процессора не будут переставлены друг с другом;

Атомарные операции

- Compare-And-Set атомарные инструкции:

```
1  int cmpxchg(volatile int *ptr, int old, int new)
2  {
3      const int stored = *ptr;
4      if (stored == old)
5          *ptr = new;
6      return stored;
7  }
```

- Пара инструкций Load Linked и Store Conditional:

```
1  int cpu = INVALID_CPU, *track = 0;
2
3  int load_linked(volatile int *ptr)
4  {
5      cpu = this_cpu();
6      return *(track = ptr);
7  }
8
9  int store_conditional(volatile int *ptr, int value)
10 {
11     if (track != ptr || cpu != this_cpu())
12         return 0;
13     cpu = INVALID_CPU;
14     track = 0;
15     *ptr = value;
16     return 1;
17 }
```

Атомарные операции

- Примеры Compare-And-Set инструкций:
 - *cmpxchg* - архитектура x86 (используйте префикс "lock", чтобы инструкция служила барьером памяти);
 - *cas* - архитектура sparc (является барьером);
- примеры Load Linked и Store Conditional:
 - *ldrex/strex* - архитектура ARM (не является барьером - требуют явного барьера перед и после, если нужно);
 - *lwarx/stwrx* - архитектура PowerPC (не являются барьером - требуют явного барьера перед и после, если нужно);

Атомарные операции

Compare-And-Set операцию можно выразить через пару Load Linked/Store Conditional:

```
1 int cmpxchg(volatile int *ptr, int old, int new)
2 {
3     do {
4         const int stored = load_linked(ptr);
5
6         if (stored != old)
7             return stored;
8     } while (!store_conditional(ptr, new));
9     return old;
10 }
```

Test-And-Set Lock

Атомарные операции позволяют реализовать взаимное исключение гораздо проще чем алгоритмы Петерсона и Лэмпорта (и другие):

```
1  #define LOCK_FREE 0
2  #define LOCK_HELD 1
3
4  struct spinlock {
5      int state;
6  };
7
8  void lock(struct spinlock *lock)
9  {
10     while (cmpxchg(&lock->state, LOCK_FREE, LOCK_HELD) != LOCK_FREE);
11 }
12
13 void unlock(struct spinlock *lock)
14 {
15     cmpxchg(&lock->state, LOCK_HELD, LOCK_FREE);
16 }
```

Test-And-Set Lock

Test-And-Set Lock в такой реализации обладает рядом недостатков:

Test-And-Set Lock

Test-And-Set Lock в такой реализации обладает рядом недостатков:

- все потоки в цикле пытаются выполнить модификацию обще переменной:
 - вспомните MESI - писать может только один;
 - много бесполезного трафика по системной шине;

Test-And-Set Lock

Test-And-Set Lock в такой реализации обладает рядом недостатков:

- все потоки в цикле пытаются выполнить модификацию обще переменной:
 - вспомните MESI - писать может только один;
 - много бесполезного трафика по системной шине;
- в такой реализации мы не можем давать никаких гарантий честности;

Test-And-Set Lock

Test-And-Set Lock в такой реализации обладает рядом недостатков:

- все потоки в цикле пытаются выполнить модификацию обще переменной:
 - вспомните MESI - писать может только один;
 - много бесполезного трафика по системной шине;
- в такой реализации мы не можем давать никаких гарантий честности;
- она не работает (приводит к deadlock-y).

Test-And-Set Lock

Рассмотрим следующий сценарий:

- в ОС запущено несколько приложений, которые используют сеть;
- на компьютере установлена *только одна* сетевая карта;
 - когда сетевая карта заканчивает обрабатывать запрос - она генерирует прерывание (в современных сетевых устройствах все не так просто);
 - обработчик прерывания берет следующий запрос из очереди;
 - очередь запросов - общий ресурс;

Test-And-Set Lock

Рассмотрим следующий сценарий:

- 1 приложение хочет поставить запрос в очередь - оно захватывает spinlock;

Test-And-Set Lock

Рассмотрим следующий сценарий:

- 1 приложение хочет поставить запрос в очередь - оно захватывает spinlock;
- 2 сетевая карта обработку запроса - генерирует прерывание;

Test-And-Set Lock

Рассмотрим следующий сценарий:

- 1 приложение хочет поставить запрос в очередь - оно захватывает spinlock;
- 2 сетевая карта обработку запроса - генерирует прерывание;
- 3 обработчик прерывания должен взять новый запрос из очереди - он пытается захватить spinlock;

Test-And-Set Lock

Рассмотрим следующий сценарий:

- ❶ приложение хочет поставить запрос в очередь - оно захватывает spinlock;
- ❷ сетевая карта обработку запроса - генерирует прерывание;
- ❸ обработчик прерывания должен взять новый запрос из очереди - он пытается захватить spinlock;
- ❹ ... и все - мы в тупике ...
 - приложение не может освободить spinlock, потому что прерывание прервало его работу;
 - обработчик прерывания не может захватить spinlock, потому что он уже захвачен приложением;

Test-And-Set Lock

- Для Test-And-Set Lock-ов нужно выключать прерывания в начале lock и включать в конце unlock.

Test-And-Set Lock

- Для Test-And-Set Lock-ов нужно выключать прерывания в начале lock и включать в конце unlock.
- Если прерывания единственный источник конкурентности (нет многоядерности), то достаточно просто выключить прерывания при входе в критическую секцию, чтобы гарантировать взаимное исключение;

Test-And-Set Lock

Уменьшить нагрузку на системную шину можно следующим простым трюком:

```
1  #define LOCK_FREE 0
2  #define LOCK_HELD 1
3
4  struct spinlock {
5      int state;
6  };
7
8  void lock(struct spinlock *lock)
9  {
10     disable_interrupts();
11     while (1) {
12         while (lock->state != LOCK_FREE) /* read only */
13             barrier();
14
15         if (cmpxchg(&lock->state, LOCK_FREE, LOCK_HELD) == LOCK_FREE)
16             break;
17     }
18 }
19
20 void unlock(struct spinlock *lock)
21 {
22     cmpxchg(&lock->state, LOCK_HELD, LOCK_FREE);
23     enable_interrupts();
24 }
```

Test-And-Set Lock

Текущая версия Test-And-Set Lock-a:

- не приводит к deadlock-ам (тут трудно что-то улучшить);
- уменьшает нагрузку на системную шину;
 - системная шина все еще довольно загружена - можно сделать лучше;
- все еще не честная ;

Test-And-Set Lock

Текущая версия Test-And-Set Lock-a:

- не приводит к deadlock-ам (тут трудно что-то улучшить);
- уменьшает нагрузку на системную шину;
 - системная шина все еще довольно загружена - можно сделать лучше;
- все еще не честная - этим и займемся.;

Ticket Lock

```
1  struct ticketpair {
2      uint16_t users;
3      uint16_t ticket;
4  } __attribute__((packed));
5
6  struct spinlock {
7      uint16_t users;
8      uint16_t ticket;
9  }
10
11 void lock(struct spinlock *lock)
12 {
13     disable_interrupts();
14     const uint16_t ticket = atomic_add_and_get(&lock->users, 1);
15
16     while (lock->ticket != ticket)
17         barrier();
18     smp_mb(); /* we don't use cmpxchg explicitly */
19 }
20
21 void unlock(struct spinlock *lock)
22 {
23     smp_mb();
24     atomic_add(&lock->ticket, 1);
25     enable_interrupts();
26 }
```

Ticket Lock

- Ticket Lock очень прост в реализации и достаточно эффективен;
- честность Ticket Lock-а не должна вызывать вопросов;
- реализация spinlock в Linux Kernel использует Ticket Lock.

Ticket Lock, обычно, хорош на практике, зачем нам еще один Lock?

- MCS Lock позволяет уменьшить нагрузку на системную шину до минимума не потеряв при этом в честности;

Ticket Lock, обычно, хорош на практике, зачем нам еще один Lock?

- MCS Lock позволяет уменьшить нагрузку на системную шину до минимума не потеряв при этом в честности;
- простая и красивая идея (если есть что-то красивое в параллельном программировании, то это MCS Lock):
 - авторы (John Mellor-Crummey and Michael L. Scott) получили приз Дейкстры в области распределенных вычислений;
 - забавный факт - Дейкстра не первым получил премию Дейкстры...

MCS Lock

Определения и реализация lock

```
1 struct mcs_lock {
2     struct mcs_lock *next;
3     int locked;
4 };
5
6 void lock(struct mcs_lock **lock, struct mcs_lock *self)
7 {
8     struct mcs_lock *tail;
9
10    self->next = NULL;
11    self->locked = 1;
12
13    tail = xchg(lock, self); /* cmpxchg without cmp part */
14
15    if (!tail)
16        return; /* there was no one in the queue */
17
18    tail->next = self;
19    barrier();
20    while (self->locked);
21    barrier();
22    smp_mb();
23 }
```

MCS Lock

Определения и реализация unlock

В отличие от других Lock-ов, unlock в MCS Lock менее тривиальная операция чем lock:

```
1 void unlock(struct mcs_lock **lock, struct mcs_lock *self)
2 {
3     struct mcs_lock *next = self->next;
4
5     if (!next) {
6         if (cmpxchg(lock, self, NULL) == node)
7             return;
8
9         while ((next = self->next) == NULL)
10             barrier();
11     }
12
13     smp_mb();
14     next->locked = 0;
15 }
```

- ❶ MCS Lock определенно сложнее Ticket Lock и его интерфейс несколько отличается от привычного
 - MCS Lock все еще достаточно прост;
 - MCS Lock был первым подобным алгоритмом, но не последним - есть варианты с нормальным интерфейсом;
- ❷ MCS Lock хранит всех претендентов в связном списке - честность вопросов не вызывает;
- ❸ нагрузка на системную шину минимальна:
 - каждый ожидающий проверяет свое и только свое поле `locked`;

Ассиметричные Lock-и

- Взаимного исключения достаточно, чтобы гарантировать корректность конкурентных программ

Ассиметричные Lock-и

- Взаимного исключения достаточно, чтобы гарантировать корректность конкурентных программ
 - *если вы его правильно реализовали;*

Ассиметричные Lock-и

- Взаимного исключения достаточно, чтобы гарантировать корректность конкурентных программ
 - *если вы его правильно реализовали;*
 - *если вы его правильно используете;*

Ассиметричные Lock-и

- Взаимного исключения достаточно, чтобы гарантировать корректность конкурентных программ
 - *если вы его правильно реализовали;*
 - *если вы его правильно используете;*
 - обычно кроме корректности нам еще важна скорость;

Ассиметричные Lock-и

- Взаимного исключения достаточно, чтобы гарантировать корректность конкурентных программ
 - *если вы его правильно реализовали;*
 - *если вы его правильно используете;*
 - обычно кроме корректности нам еще важна скорость;
- взаимное исключение можно оптимизировать под специфичную нагрузку:
 - Read/Write Lock - читатели могут работать параллельно (писатели ждут);

Ассиметричные Lock-и

- Взаимного исключения достаточно, чтобы гарантировать корректность конкурентных программ
 - *если вы его правильно реализовали;*
 - *если вы его правильно используете;*
 - обычно кроме корректности нам еще важна скорость;
- взаимное исключение можно оптимизировать под специфичную нагрузку:
 - Read/Write Lock - читатели могут работать параллельно (писатели ждут);
- можно использовать совершенно другие подходы:
 - атомарный снэпшот (seqlock);
 - неблокирующая синхронизация (obstruction/lock/wait free);

Атомарный снимок

- Есть некоторые данные
 - *не списочная структура*, т. е. просто непрерывный участок в памяти;

Атомарный снимок

- Есть некоторые данные
 - *не списочная структура*, т. е. просто непрерывный участок в памяти;
- параллельное изменение данных не допускается:
 - чтобы изменить данные нужно захватить Lock;
 - модификации данных довольно редкие;

Атомарный снимок

- Есть некоторые данные
 - *не списочная структура*, т. е. просто непрерывный участок в памяти;
- параллельное изменение данных не допускается:
 - чтобы изменить данные нужно захватить Lock;
 - модификации данных довольно редкие;
- необходимо обеспечить эффективное чтение этих данных:
 - несколько читателей могут работать параллельно;
 - нужно обезопасить читателей от писателей изменяющих данные;

Атомарный снимок

Определения и чтение

```
1 struct seqlock {
2     struct spinlock lock;
3     unsigned sequence;
4 };
5
6 unsigned read_seqbegin(const struct seqlock *seq)
7 {
8     unsigned sequence = seq->sequence;
9
10    smp_rmb();
11    return ret & ~1u;
12 }
13
14 int read_seqretry(const struct seqlock *seq, unsigned sequence)
15 {
16     smp_rmb();
17     return (seq->sequence != sequence);
18 }
```

Атомарный снимок

Запись

```
1 void write_seqbegin(struct seqlock *seq)
2 {
3     lock(&seq->lock);
4     seq->sequence++;
5     smp_wmb();
6 }
7
8 void write_seqend(struct seqlock *seq)
9 {
10    smp_wmb();
11    seq->sequence++;
12    unlock(&seq->lock);
13 }
```

Атомарный снимок

- seqlock *при правильном использовании* фактически бесплатен для читателей;
- seqlock не откладывает доступ писателям, пока есть читатели в отличие от Read/Write Lock-ов;
 - писатель имеет приоритет над читателями, поэтому писателей должно быть мало;
 - чем больше структура - тем реже она должна изменяться;
- почти не пригоден для защиты списочных структур данных - применение очень ограничено;