

Operating Systems

Threads Synchronization

Me

February 27, 2016

- 1 Конкурентное исполнение. Состояние Гонки.
- 2 Взаимное исключение. Алгоритм Петерсона.
- 3 Честное взаимное исключение. Алгоритм пекарни.
- 4 Переупорядочивание. 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 Петерсона 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.

Переупорядочивание

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

Переупорядочивание

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

- компилятору *разрешено* переупорядочивать 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

Переупорядочивание

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

- компилятору *разрешено* переупорядочивать 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 буффер 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 переупорядочивать обращения к `volatile` данным, если они *разделены точкой следования*;
 - компилятор может переупорядочивать 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

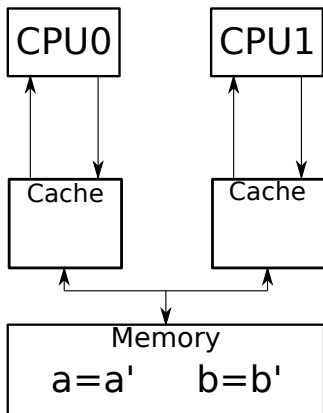


Figure : Cache Incoherency

Когерентность процессорных кешей

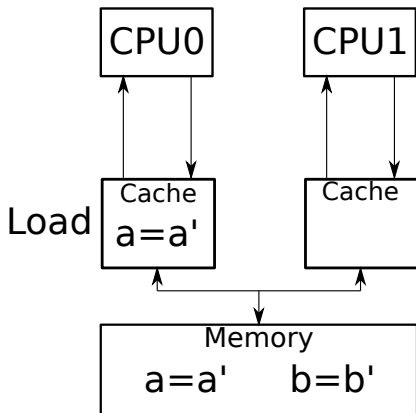


Figure : Cache Incoherency

Когерентность процессорных кешей

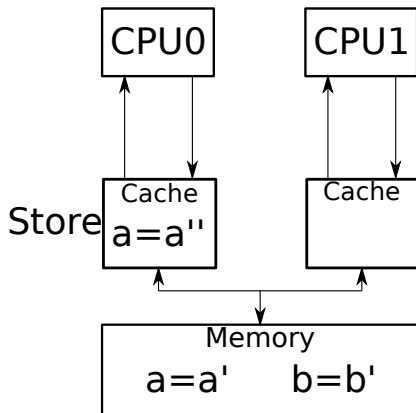


Figure : Cache Incoherency

Когерентность процессорных кешей

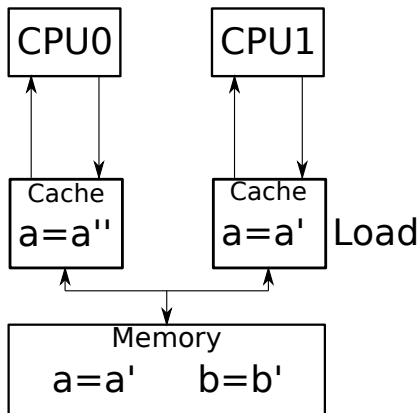


Figure : Cache Incoherency

Когерентность процессорных кешей

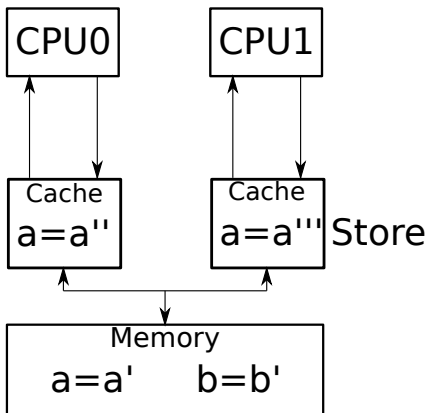


Figure : Cache Incoherency

Когерентность процессорных кешей

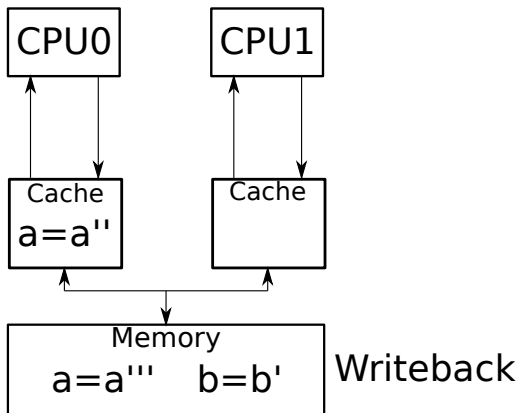


Figure : Cache Incoherency

Когерентность процессорных кешей

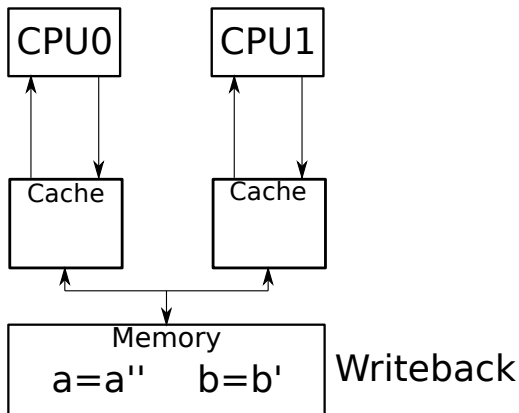


Figure : Cache Incoherency

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 широкоизвестный 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;
- если никто не держит данные в кеше, то мы получаем их от контроллера памяти;