

Operating Systems

Advanced Synchronization

Me

March 15, 2016

- 1 Неблокирующая синхронизация.
Obstruction/Lock/Wait freedom.
- 2 Пример: lock free stack.
- 3 Проблема АВА. Безопасное освобождение памяти.
- 4 Tagged Pointer. Bounded lock free stack;
- 5 Hazzard Pointers.

Неблокирующая синхронизация

Зачем нам неблокирующая синхронизация?

- deadlock-и иногда случаются
 - возможно ошибка в алгоритме (такое случается, но нам не особо интересно);
 - поток исполнения может упасть в критической секции (хотя мы и опирались явно на обратное);
- инверсия приоритетов:
 - менее приоритетный поток держит Lock, и не может ее отпустить, потому что ему не дают процессорное время;

Неблокирующая синхронизация

Можно ли обходиться без Lock-ов?

- да, конечно, у нас ведь есть `strxchg` - нужно просто вызывать его в правильном порядке;

Неблокирующая синхронизация

Можно ли обходиться без Lock-ов?

- все, конечно, не так просто:
 - правильный порядок не всегда очевиден;
 - не каждый порядок можно считать неблокирующим - нам нужно более формальное определение;
 - придется решать новые проблемы, которых не было при использовании Lock-ов;
 - результат может быть непрактичным (медленным).

Definition

Реализация алгоритма/структуры данных obstruction free, если начиная с любого достижимого состояния, любой поток исполнения может завершить выполнение задачи за конечное количество шагов, если *все другие потоки остановятся*.

Definition

Реализация lock free, если *один из потоков* (не известно какой) гарантированно завершает выполнение своей задачи за конечное число шагов, *независимо от других потоков*.

Definition

Реализация wait free, если *каждый поток* завершает свою операцию за конечное количество шагов *независимо от других потоков*.

Lock free stack

```
1  template <typename T>
2  class Stack {
3      struct StackNode {
4          StackNode *next;
5          T *data;
6
7          StackNode(T *data) : next(nullptr), data(data) { }
8      };
9
10     std::atomic<StackNode *> head;
11
12 public:
13     Stack();
14     ~Stack();
15
16     void push(T *value);
17     T *pop();
18 };
```

Lock free stack

```
1         assert(head.load(std::memory_order_relaxed) == nullptr);  
2     }  
3  
4     template <typename T>  
5     void Stack<T>::push(T *value)  
6     {  
7         StackNode *node = new StackNode(value);  
8         StackNode *expected;  
9  
10        do {  
11            expected = head.load(std::memory_order_relaxed);
```

Lock free stack

```
1         } while (!head.compare_exchange_strong(expected, node));
2     }
3
4     template <typename T>
5     T *Stack<T>::pop()
6     {
7         StackNode *node;
8         StackNode *next;
9
10        do {
11            node = head.load(std::memory_order_relaxed);
12            if (!node)
13                return nullptr;
14            next = node->next;
15        } while (!head.compare_exchange_strong(node, next));
16
17        T *data = node->data;
```

Lock free stack

К сожалению эта реализация:

- не obstruction/lock/wait free (возможно);

Lock free stack

К сожалению эта реализация:

- не obstruction/lock/wait free (возможно);
 - чтобы реализация была lock free, она не должна использовать не lock free примитивов как минимум;

Lock free stack

К сожалению эта реализация:

- не obstruction/lock/wait free (возможно);
 - чтобы реализация была lock free, она не должна использовать не lock free примитивов как минимум;
 - является ли вызов new lock free?

Lock free stack

К сожалению эта реализация:

- не obstruction/lock/wait free (возможно);
 - чтобы реализация была lock free, она не должна использовать не lock free примитивов как минимум;
 - является ли вызов new lock free?
- не корректна:

Lock free stack

К сожалению эта реализация:

- не obstruction/lock/wait free (возможно);
 - чтобы реализация была lock free, она не должна использовать не lock free примитивов как минимум;
 - является ли вызов new lock free?
- не корректна:
 - обращается к памяти после освобождения;

Lock free stack

К сожалению эта реализация:

- не obstruction/lock/wait free (возможно);
 - чтобы реализация была lock free, она не должна использовать не lock free примитивов как минимум;
 - является ли вызов new lock free?
- не корректна:
 - обращается к памяти после освобождения;
 - неправильно использует Compare-And-Set примитив для синхронизации;

Безопасное освобождение памяти

```
1 node = this->head.load(std::memory_order_relaxed);  
2 if (!node)  
3     return nullptr;  
4 next = node->next;
```

Имеем ли мы право разыменовывать указатель `node`?

- несколько вызовов `pop` могут выполняться параллельно;
- один из них мог освободить память, после того как мы прочитали `node` и до того, как мы прочитали `node->next`;

Проблема АВА

```
1 node = this->head.load(std::memory_order_relaxed);  
2 if (!node)  
3     return nullptr;  
4 next = node->next;
```

Гораздо более тонкая неприятность:

- `compare_exchange_strong` успешен, если значение хранимое в атомарной переменной *побитово совпадает* с ожидаемым значением (это упрощение);

Проблема АВА

```
1 node = this->head.load(std::memory_order_relaxed);  
2 if (!node)  
3     return nullptr;  
4 next = node->next;
```

Гораздо более тонкая неприятность:

- `compare_exchange_strong` успешен, если значение хранимое в атомарной переменной *побитово совпадает* с ожидаемым значением (это упрощение);
- нам нужно, чтобы `compare_exchange_strong` был успешен, если значение в атомарной переменной *не менялось* с момента, когда мы сделали `load` (как в LL/SC);

Проблема АВА

- У нас есть несколько потоков работающих со стеком параллельно:
 - Thread0 выполняют операцию pop;
 - Thread1 выполняет операцию pop и затем сразу операцию push;
 - все остальные потоки каким-то образом модифицируют стек;

Проблема АВА

```
1 node = this->head.load(std::memory_order_relaxed);  
2 if (!node)  
3     return nullptr;  
4 next = node->next;
```

- Thread0 загрузил голову стека (пусть ее адрес Addr, т. е. `node == Addr`);
- Thread0 проверил, что она не равна `nullptr`, т. е. стек не пуст;
- Thread0 запоминает указатель на следующий элемент стека;
- после чего Thread0 был снят с процессора;

Проблема АВА

- Thread1 выполняет операцию pop полностью:
 - удаляет голову стека из списка;
 - освобождает память занятую головой списка, т. е. Addr теперь свободный адрес;
- Thread1 снимается с процессора и управление получают другие потоки, которые изменяют стек;
- Thread1 снова получает управление и выполняет операцию push полностью:
 - Thread1 должен алоцировать новую голову стека, в частности Addr - свободный адрес;
 - Thread1 добавляет новую голову с адресом Addr в стек;

Проблема АВА

```
1 do {  
2     ...  
3 } while (!this->head.compare_exchange_strong(node, next));
```

- Thread0 вновь получает процессор и продолжает исполнение операции pop;
- Thread0 выполняет `compare_exchange_strong` - операция успешна, потому что после Thread1 адрес головы стека вновь стал Addr;

Проблема АВА

```
1 do {  
2     ...  
3 } while (!this->head.compare_exchange_strong(node, next));
```

- Thread0 вновь получает процессор и продолжает исполнение операции pop;
- Thread0 выполняет `compare_exchange_strong` - операция успешна, потому что после Thread1 адрес головы стека вновь стал Addr;
- но поле `next` в голове стека могло измениться!

Не все биты указателя реально используются:

- например, данные в динамической памяти зачастую алоцируются выровненными на границу 2^i , где i обычно небольшое число (4-5);
- в x86-64, под адресацию используется только 48 бит, оставшиеся 16 либо 0 либо 1 (канонический адрес);
- мы можем создавать свои собственные "указатели" разного размера (увидим дальше в примере);

Не все биты указателя реально используются:

- например, данные в динамической памяти зачастую алоцируются выровненными на границу 2^i , где i обычно небольшое число (4-5);
- в x86-64, под адресацию используется только 48 бит, оставшиеся 16 либо 0 либо 1 (канонический адрес);
- мы можем создавать свои собственные "указатели" разного размера (увидим дальше в примере);
- мы можем использовать эти неиспользуемые биты!

Tagged Pointer

```
1 typedef std::uint16_t   StackNodePtr ;  
2 typedef std::uint16_t   StackNodeTag ;  
3 typedef std::uint32_t   StackHeadPtr ;
```

Заведем свой "внутренний" указатель:

- StackNodePtr - 16-битное число, которое будет выступать в качестве указателя элемента стека;
 - что вообще можно адресовать 16-битным числом?

Tagged Pointer

```
1 typedef std::uint16_t   StackNodePtr ;  
2 typedef std::uint16_t   StackNodeTag ;  
3 typedef std::uint32_t   StackHeadPtr ;
```

Заведем свой "внутренний" указатель:

- StackNodePtr - 16-битное число, которое будет выступать в качестве указателя элемента стека;
 - что вообще можно адресовать 16-битным числом?
 - все что угодно, если этих объектов не больше чем 2^{16}

Tagged Pointer

```
1 typedef std::uint16_t    StackNodePtr ;  
2 typedef std::uint16_t    StackNodeTag ;  
3 typedef std::uint32_t    StackHeadPtr ;
```

Заведем свой "внутренний" указатель:

- StackNodePtr - 16-битное число, которое будет выступать в качестве указателя элемента стека;
 - что вообще можно адресовать 16-битным числом?
 - все что угодно, если этих объектов не больше чем 2^{16}
- StackNodeTag - 16-битное число, служебная информация, которую мы храним рядом с указателем элемента;

Tagged Pointer

```
1 typedef std::uint16_t StackNodePtr;  
2 typedef std::uint16_t StackNodeTag;  
3 typedef std::uint32_t StackHeadPtr;
```

Заведем свой "внутренний" указатель:

- StackNodePtr - 16-битное число, которое будет выступать в качестве указателя элемента стека;
 - что вообще можно адресовать 16-битным числом?
 - все что угодно, если этих объектов не больше чем 2^{16}
- StackNodeTag - 16-битное число, служебная информация, которую мы храним рядом с указателем элемента;
- StackHeadPtr - объединение StackNodePtr и StackNodeTag в одно 32-битное число.

Tagged Pointer

Функции для работы с StackNodePtr, StackNodeTag и StackHeadPtr:

```
1 static StackHeadPtr createPtr(StackNodePtr ptr, StackNodeTag tag)
2 { return ptr | ((StackHeadPtr)tag << 16); }
3
4 static StackNodePtr getPtr(StackHeadPtr ptr)
5 { return ptr & 0xffff; }
6
7 static StackNodeTag getTag(StackHeadPtr ptr)
8 { return ptr >> 16; }
```


Tagged Pointer

```
1 struct StackNode {  
2     StackNodePtr next;  
3     T *data;  
4  
5     StackNode(T *data = nullptr) : next(0), data(data) { }  
6 };  
7  
8 StackNode * pool;  
9 std::atomic<StackHeadPtr> head;  
10 std::atomic<StackHeadPtr> free;
```

- Изменим определение полей нашего стека, чтобы они использовали StackHeadPtr и StackNodePtr;
- Чтобы StackNodePtr "указывал" на реальный объект в памяти заведем массив pool
 - StackNodePtr будет индексом в этом массиве;
- Для аллоции и освобождения элементов стека динамически используем список free;

Аллокация узлов стека

```
1 StackNodePtr allocatePtr()  
2 {  
3     StackHeadPtr head, next;  
4     StackNodePtr ptr;  
5  
6     do {  
7         head = free.load(std::memory_order_relaxed);  
8         ptr = getPtr(head);  
9  
10        if (!ptr)  
11            continue;  
12  
13        StackNode *node = getNode(ptr); // &pool[ptr]  
14        next = createPtr(node->next, getTag(head) + 1);  
15    } while (!free.compare_exchange_strong(head, next));  
16  
17    return ptr;  
18 }
```

Освобождение узлов стека

```
1 void freePtr(StackNodePtr ptr)
2 {
3     StackHeadPtr head, new_head;
4     StackNode *node = getNode(ptr);
5
6     do {
7         head = free.load(std::memory_order_relaxed);
8         node->next = getPtr(head);
9         new_head = createPtr(next, getTag(head) + 1);
10    } while (!free.compare_exchange_strong(head, new_head));
11 }
```

Реализация push

```
1  template <typename T>
2  void Stack<T>::push(T *value)
3  {
4      StackHeadPtr old_head, new_head;
5      StackNodePtr ptr = allocatePtr();
6      StackNode *node = getNode(ptr);
7      node->data = value;
8      do {
9          old_head = head.load(std::memory_order_relaxed);
10         new_head = createPtr(ptr, getTag(old_head) + 1);
11         node->next = getPtr(old_head);
12     } while (!head.compare_exchange_strong(old_head, new_head));
13 }
```

Реализация pop

```
1  template <typename T>
2  T *Stack<T>::pop()
3  {
4      StackHeadPtr old_head, new_head;
5      StackNodePtr ptr;
6      StackNode *node;
7
8      do {
9          old_head = head.load(std::memory_order_relaxed);
10         ptr = getPtr(old_head);
11         if (!ptr)
12             return nullptr;
13         node = getNode(ptr);
14         new_head = createPtr(node->next, getTag(old_head) + 1);
15     } while (!head.compare_exchange_strong(old_head, new_head));
16
17     T *data = node->data;
18     freePtr(ptr);
19     return data;
20 }
```

Hazard Pointer

- Tagged Pointer - простая и достаточно эффективная техника, но не универсальная;
- Мы должны знать какого размера Tag-а нам точно достаточно:
 - если поток заснул на долго, то Tag может переполниться и он этого не заметит;
 - работа с большим Tag-ом может вообще не быть атомарной;
- Возьмем Tagged Pointer на вооружение, но поищем альтернативу (тем более, что они есть);

Hazard Pointer

Hazard Pointer позволяет отметить указатель, как "небезопасный" и тем самым отложить его освобождение

- оригинальная статья: "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects", Maged M. Michael;
- Hazard Pointer популярная универсальная техника, но мы рассмотрим только простую (и не универсальную) реализацию.

Hazard Pointer

Основная идея:

- У каждого потока есть небольшой набор Hazard Pointer-ов, в которые он сохраняет указатели на используемые объекты (которые нельзя удалять);
- Писать в Hazard Pointer может только поток-владелец, но читать могут все;
- Перед тем как освободить память проверим все Hazard Pointer-ы всех потоков.

Hazard Pointer

Перед тем как рассмотреть реализацию, посмотрим как мы будем ее использовать, начнем с простого - реализация push:

```
1  template <typename T>
2  void Stack<T>::push(T *value)
3  {
4      StackNode *node = new StackNode(value);
5      StackNode *expected;
6
7      do {
8          expected = this->head.load(std::memory_order_relaxed);
9          node->next = expected;
10     } while (!this->head.compare_exchange_strong(expected, node));
11 }
```

Hazard Pointer

Тепрь реализация pop:

```
1  template <typename T>
2  T *Stack<T>::pop()
3  {
4      HazardPtr<StackNode> hp(acquireRawHazardPtr(0));
5      StackNode *node;
6      StackNode *next;
7
8      do {
9          node = head.load();
10         do {
11             if (!node)
12                 return nullptr;
13             hp.set(node);
14         } while ((node = head.load()) != hp.get());
15
16         next = node->next;
17     } while (!head.compare_exchange_strong(node, next));
18
19     hp.set(nullptr);
20     T *data = node->data;
21     releasePtr(node);
22     return data;
23 }
```

Hazard Pointer

Наша реализация будет очень простой потому что мы допустим несколько упрощений:

- максимальное количество потоков нам известно заранее;
 - полноценная реализация должна учитывать, что потоки могут создаваться и завершаться;
- максимальное количество Hazard Pointer-ов на один поток нам известно заранее
 - обычно потокам не нужно много Hazard Pointer-ов одновременно (3-4), поэтому это довольно слабое ограничение;
- другими словами, мы можем статически алоцировать все Hazard Pointer-ы.

Hazard Pointer

```
1 static const std::size_t MAX_THREAD_COUNT = 1024;
2 static const std::size_t MAX_THREADPTR_COUNT = 16;
3 static const std::size_t MAX_HAZARD_COUNT = MAX_THREAD_COUNT *
    ↳ MAX_THREADPTR_COUNT;
4
5 typedef void * raw_ptr_t;
6 static std::atomic<raw_ptr_t> hazard[MAX_HAZARD_COUNT];
7
8 std::atomic<raw_ptr_t> *acquireRawHazardPtr(int idx)
9 {
10     assert(idx < MAX_THREADPTR_COUNT);
11     const int tid = getThreadId();
12     return &hazard[tid * MAX_THREADPTR_COUNT + idx];
13 }
```

- Hazard Pointer - глобальный атомарный указатель
 - каждый поток может получить один из своих Hazard Pointer-ов по индексу (спасибо нашему упрощению);
 - чтобы обезопасить указатель нужно сохранить его в атомарный указатель;
 - когда вы закончите работать с указателем запишите туда 0;

Hazard Pointer

```
1 void releaseRawPtr(raw_ptr_t ptr, std::function<void(raw_ptr_t)> deleter  
   ↪ )  
2 {  
3     static thread_local std::vector<RetireNode> retireList;  
4     static const size_t RETIRE_THRESHOLD = 10; // should be much bigger  
5  
6     RetireNode node = { deleter, ptr };  
7     retireList.push_back(node);  
8     if (retireList.size() > RETIRE_THRESHOLD)  
9         collectGarbage(retireList);  
10 }
```

- Каждый поток поддерживает список ожидающих удаления элементов;
- Если список стал слишком большим - запускаем сборку "мусора";

Hazard Pointer

```
1 static void collectGarbage(std::vector<RetireNode> &retireList)
2 {
3     std::set<raw_ptr_t> plist;
4     for (std::size_t i = 0; i != MAX_HAZARD_COUNT; ++i) {
5         raw_ptr_t ptr = hazard[i].load();
6
7         if (ptr != nullptr)
8             plist.insert(ptr);
9     }
10
11     std::vector<RetireNode> retire;
12     retire.swap(retireList);
13     for (std::size_t i = 0; i != retire.size(); ++i) {
14         if (plist.count(retire[i].ptr))
15             retireList.push_back(retire[i]);
16         else
17             retire[i].deleter(retire[i].ptr);
18     }
19 }
```

- Для каждого элемента в списке проверим, защищен ли он Hazard Pointer-ом:
 - если да, то вернем указатель назад в список ожидания;
 - если нет, то его можно безопасно удалять;

Финальные замечания

- Lockless алгоритмы не редко трудны для реализации и понимания
 - используйте Lock-и по умолчанию, и Lockless, когда Lock-и стали проблемой;
- Lockless алгоритмы не производительнее и не масштабируются лучше чем Lock-based:
 - Lockless алгоритмы предоставляют гарантии прогресса, а не производительности;
 - на производительности и масштабируемости в первую очередь сказывается конкуренция за общие ресурсы;