



ТЕОРИЯ И ПРАКТИКА МНОГОПОТОЧНОГО ПРОГРАММИРОВАНИЯ

Тема 13

Неблокирующие алгоритмы и их проблемы.

Д.ф.-м.н., профессор А.Г. Тормасов

Базовая кафедра «Теоретическая и Прикладная Информатика», МФТИ

Тема

- Краткий обзор некоторых неблокирующих алгоритмов, применяемых на практике
- Некоторые алгоритмы и подходы, созданные в последнее время, но не получившие пока распространения

Lockless структуры

- Их создание трудно
- Сложно верифицировать полученный код
- Иногда их использование невозможно
 - Вообще невозможно математически
 - Возможно, но полученное решение содержит слишком много памяти и/или команд, блокирующих шину памяти
- Почти всегда используются компромиссы

Некоторые идеи

- Неблокирующие алгоритмы – слабый вариант свободы от ожидания
- Оптимистически считают, что конфликты редки и их надо считать скорее исключением, чем правилом
- Но мы должны уверенно детектировать наличие конфликта, и обычно просто рестартовать операцию

Некоторые идеи

- Для обновления разделяемых данных неблокирующие алгоритмы используют чаще всего следующий алгоритм:
 - Подготовить копию (возможно частичную) объекта, который вы хотите обновить
 - Обновить копию
 - Заменить основные данные на сделанную копию атомарным образом, например, заменив указатель на нее через CAS операцию
 - Если вдруг произошел конфликт, то повторить с начала, удалив текущую копию
- Подразумеваем, что объект сконструирован как динамический набор узлов, соединённых указателями (есть указатель для каждого объекта, через который осуществляется исключительный доступ).

Некоторые идеи

- Еще одна идея реализации – не требовать указателей и их атомарной замены, а использовать порядок чтения-записи
- Пример – разделяемые буферы, алгоритм Лампорта
 - L. Lamport, “Concurrent reading and writing”
Communications of the ACM, vol. 20, no. 11, pp. 806–811,
Nov. 1977

Некоторые идеи

- Для свободной от ожидания реализации часто используются сложные алгоритмы
- Одна из идей – копирование объекта в разные буферы, и перенаправление соисполняемых операций в разные буферы для избежания конфликтов.
 - Буфер с наиболее последним значением обычно считается актуальным и активируется через сменный указатель (так как уже было описано)

Некоторые идеи

- Еще одна идея – помощь в работе другим методам
 1. Каждая операция, назовем ее операция_i, сначала анонсирует некоторую информацию о том, что она собирается делать с разделяемым объектом в некой глобальной структуре.
 2. Затем, эта глобальная структура на другие операции, которые анонсированы и не завершены, например, из за прерывания. Анонсированные операции затем выполняются операцией_i, обычно в несколько шагов, которые каким то образом используют атомарные примитивы для гарантий.
 3. Когда все анонсированные операции получили помощь, то сама уже операция_i выполняется примерно так же, как когда она помогала другим операциям. Так как все остальные соисполняющиеся операции так же следуют общей схеме помощи, то может оказаться что и операция_i помогает сама себе, то есть процедура помощи фактически работает рекурсивно.
- Этот метод требует, чтобы операция была разбита на несколько шагов, удовлетворяющих определенным требованиям, нужным для гарантии непротиворечивости между соисполняемыми операциями, которые пытаются помочь в выполнении операции.

Ядро Linux

- Структуры, используемые в ядре Linux (2.6)
 - krefs (lockless counters – самый типичный пример использования низкоуровневых счетчиков)
 - seqlocks (подход, отчасти аналогичный транзакционной памяти по идеологии)
 - Ring buffer (kfifo.h)
 - RCU (radix tree в кеше страниц файловой системы)

krefs

- Самая стандартная схема использования атомарных инструкций
- Применяется для подсчета ссылок на объект
- По удалению последней ссылки объект обычно удаляется

krefs

ИСПОЛЬЗОВАНИЕ

```
struct foo {...
    struct kref kref;
};
// инициализация вместе со структурой
struct foo *foo;
foo = kmalloc(sizeof(*foo), GFP_KERNEL);
kref_init(&foo->kref, foo_release);
// увеличение-уменьшение счетчика
kref_get(&foo->kref);
kref_put(&foo->kref);
// удаление
void foo_release(struct kref *kref)
{ struct foo *foo;
  foo=container_of(foo, struct foo, kref);
  kfree(foo);
}
```

реализация

```
void kref_init(struct kref *kref,
              void (*release), (struct kref *kref))
{ atomic_set(&kref->refcount, 1);
  kref->release = release;
}

struct kref *kref_get(struct kref *kref) {
  atomic_inc(&kref->refcount);
  return kref;
}

...
down (&disconnect_sem);

...
kref_put(&dev->kref);
up(&disconnect_sem);
```


seqlocks

- sequential lock
- Придуман для следующей схемы:
 - предохраняемый разделяемый ресурс мал и прост
 - Обычно не содержит указателей
 - часто используется на чтение
 - запись редка, но тоже должна быть быстрой
- Читатель получает полный быстрый доступ к ресурсу без ожидания
 - Плата – требует проверки условия коллизии с писателем на выходе (счетчик)
 - Повтор операции чтения, если коллизия с писателем
- Писатель должен получить исключительный доступ к данным через явную блокировку

seqlocks

```
// вариация на тему read-write lock

// использование последовательного блока
// читатель всегда перед работой получает некое
// целое значение seq

unsigned int seq;

do {

    seq = read_seqbegin(&the_lock);

    /* Do what you need to do */

// на выходе мы всегда проверяем значение seq
} while read_seqretry(&the_lock, seq);
// выходим только если оно не изменилось

// если изменилось, то писатель уже начал работу
// и надо повторить операцию чтения
```

```
// писатель всегда использует
// традиционный блок «от себе подобных»
// никогда не ждет читателя! То есть, быстр
void write_seqlock(seqlock_t *lock);
// Или
void write_tryseqlock(seqlock_t *lock);
```

- Более эффективен когда много читателей и мало писателей
- Хотя может быть «активная блокировка»
- Писатель может инвалидировать значение указателя если он содержится в разделяемой области, обычно не применяется
 - рекомендуется использовать RCU
- нельзя отлаживать в дебагере, так как исполнение кода почти всегда вызовет retry !

RCU

- Оптимизировано для частого чтения и редкой записи
- Защищенный ресурс должен быть доступен только через один указатель, и все ссылки на него должны обрабатываться атомарно
- Для изменения данных писатель делает себе полную копию, меняет копию и атомарно меняет указатель на свою копию
- Данные освобождаются после удаления ссылок на них
- В невытесняющем ядре накладные расходы на чтение – 0! Даже барьер памяти не нужен

RCU

ЧТЕНИЕ

```
struct my_stuff *stuff;  
  
// быстр и ничего не ждет  
rcu_read_lock( );  
stuff =  
find_the_stuff(args...);  
do_something_with(stuff);  
rcu_read_unlock( );
```

ЗАПИСЬ

- Выделить память для копии
- Скопировать
- Изменить
- Заменить указатель (+барьер)
- Дождаться выполнения кода на каждом процессоре
 - Или сделать refcount
- Освободить память (оригинал)

Неблокирующий стек Treiber

добавление

```
struct Node {
    value_t data;
    Node *next;
};
Node *S;

void push(value_t v)
{
    Node *t, *x = new Node();
    x->data = v;
    do { t = S;
        x->next = t;
    } while(!CAS(&S,t,x));
}
```

удаление

```
void init()
{
    S = NULL;
}
value_t pop()
{
    Node *t, *x;
    do {
        t = S;
        if (t == NULL)
            return EMPTY;
        x = t->next;
    } while(!CAS(&S,t,x));
    return t->data;
}
```

Неблокирующий стек Treiber

- Функция `init` инициализирует структуры.
- Операция `push`:
 1. Выделяется память нового узла `x`;
 2. Читаем текущее значение указателя вершины стека `S`;
 3. Устанавливает поле `next` нового узла в считанное значение `S`;
 4. Атомарно обновляет вершину стека адресом нового узла `x`.
- Если значение указателя между 2 и 4 меняется и не установлено в его начальное значение, то операция `CAS` не выполняется и операция запускается повторно.
- Операция `pop` работает аналогично.
- Алгоритм свободен от блокировок, но не от ожидания

Неблокирующая очередь

- Предложена Михаэлем и Скоттом «по мотивам» стека Трибера, как односвязный список с указателям на Head и Tail.
- Head всегда указывает на пустой узел, Tail – на последний или следующий за последним узел
- Использует CAS и счетчик (от ABA)
- Dequeue работает так, что Tail не указывает ни на удаляемый узел, ни на его предшественника (удаленный узел можно повторно использовать)
- Для обеспечения согласованности используется последовательность операций чтения, перезачитывающих значения (убедиться, что они неизменны)
- Алгоритм линеаризуем (есть моменты изменения состояния, и состояние очереди никогда не может быть неправильно прочитано, нет «переходных процессов»)

Неблокирующая очередь

структуры

```
struct Pointer {
    Node *ptr;
    unsigned int counter;
};

struct Node {
    value_t value;
    Pointer next;
};

struct Queue {
    Pointer Head;
    Pointer Tail;
};

Queue *;
```

инициализация

```
void init(Queue *q)
{
    Node *x = new Node();
    x->next.ptr = NULL;
    q->Head = <x,0>;
    q->Tail = <x,0>; // один узел
}
```

Неблокирующая очередь

добавление

```
void enqueue(Queue *q,
             value_t value)
{
    Node *x = new Node();
    x->value = value;
    x->next.ptr = NULL;
    while(1)
    {
        // читаем ptr+cnt вместе!
        Pointer t = q->Tail;
        Pointer n = t.ptr->next;
```

добавление

```
        if (t == q->Tail) // согласованы?
        {
            // t на последний узел?
            if (n.ptr == NULL)
            {
                // пытаемся вставить в конец
                if (CAS(&t.ptr->next, n,
                       <x,n.count+1>))
                    break; // закончили
            } else // t - не последний узел
                CAS(&q->Tail, t, // след. узел
                   <n.ptr,t.count+1>);
        }
        // добавленный узел - в Tail
        CAS(&q->Tail,t,<x,t.counter+1>);
    }
}
```


Неблокирующая очередь

удаление

```
BOOL dequeue(Queue *q,
             value_t *pvalue)
{
    while(1) {
        // читаем значения
        Pointer t=q->Tail, h=q->Head,
              n=h->next;
        if (h == q->Head) // согласованы?
            // пусто или хвост сзади?
            if (h.ptr == t.ptr)
                {
                    if (n.ptr == NULL)
                        // очередь пуста
                        return FALSE;
                }
    }
}
```

удаление

```
        // хвост сзади, двигаем вперед
        CAS(&q->Tail, t,
           <n.ptr,t.count+1>));
    } else // на хвост не смотрим
    { // читаем здесь, иначе другой
        // dequeue удалит следующий
        узел
        *pvalue = n.ptr->value;
        if (CAS(&q->Head, h,
              <n.ptr,h.count+1>))
            break; // сл. узел для
        головы
    }
} // while
free(h.ptr);
return TRUE;
}
```

Без атомарных инструкций

- Кольцевой FIFO буфер с SWSR
- RCU с SWMR
 - Читатель и писатель свободны от ожидания
 - До момента рекламации памяти писателем
- Алгоритм Деккера (только для 2 потоков, свободен от блокировки и зависания, требует барьеров)
- Fastflow – базируется на SRSW очередях, реализует произвольную сеть передачи информации

Алгоритм Деккера

Поток 1

```
boolean flag[2] = {false, false};
int turn = 0; // или 1

flag[0] = true;
while (flag[1] == true) {
    if (turn != 0) {
        flag[0] = false;
        while (turn != 0)
            ;
        flag[0] = true;
    }
}
// критическая секция
...
turn = 1;
flag[0] = false;
// конец критической секции
```

Поток 2

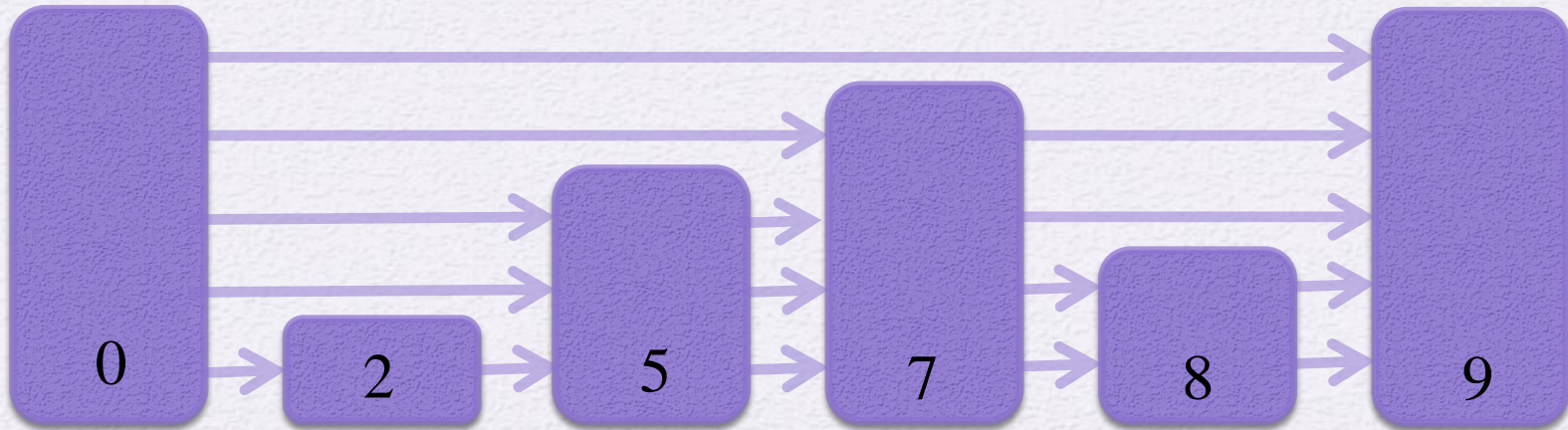
```
flag[1] = true;
while (flag[0] == true) {
    if (turn != 1) {
        flag[1] = false;
        while (turn != 1)
            ;
        flag[1] = true;
    }
}
// критическая секция
...
turn = 0;
flag[0] = false;
// конец критической секции
```


список с пропусками skip list

- Типовые поисковые структуры с нагрузкой типа 90% - lookup, 9% - add, 1% - del
- Аналог – балансированные деревья (AVL, красно-черные и др)
 - Общая проблема ребалансировки после add/del – это глобальная плохо параллелизуемая операция
- Список с пропусками, предложен в 90-х годах
 - Вероятностная структура с логарифмическим поиском
 - $O(\log(n))$ в среднем, $O(n)$ в худшем
 - Нет глобальной ребалансировки
- Далее рассмотрим пример из книги Херлихая и Шафита

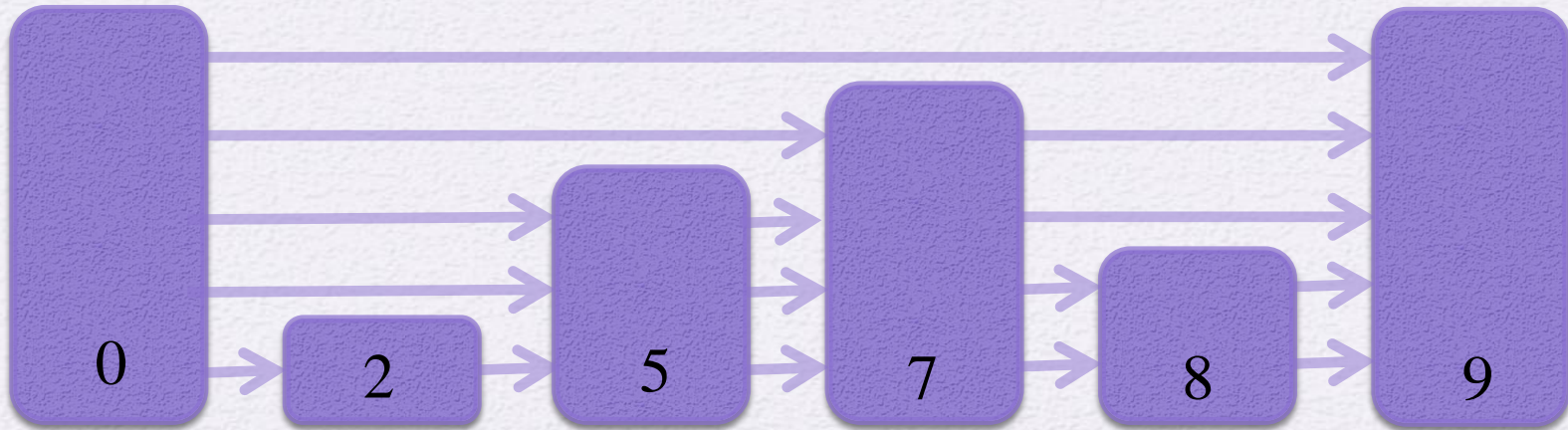
список с пропусками skip list

- Многоуровневый дублирующий набор списков
 - Список выше уровнем есть часть низкоуровневого
 - Они все упорядочены



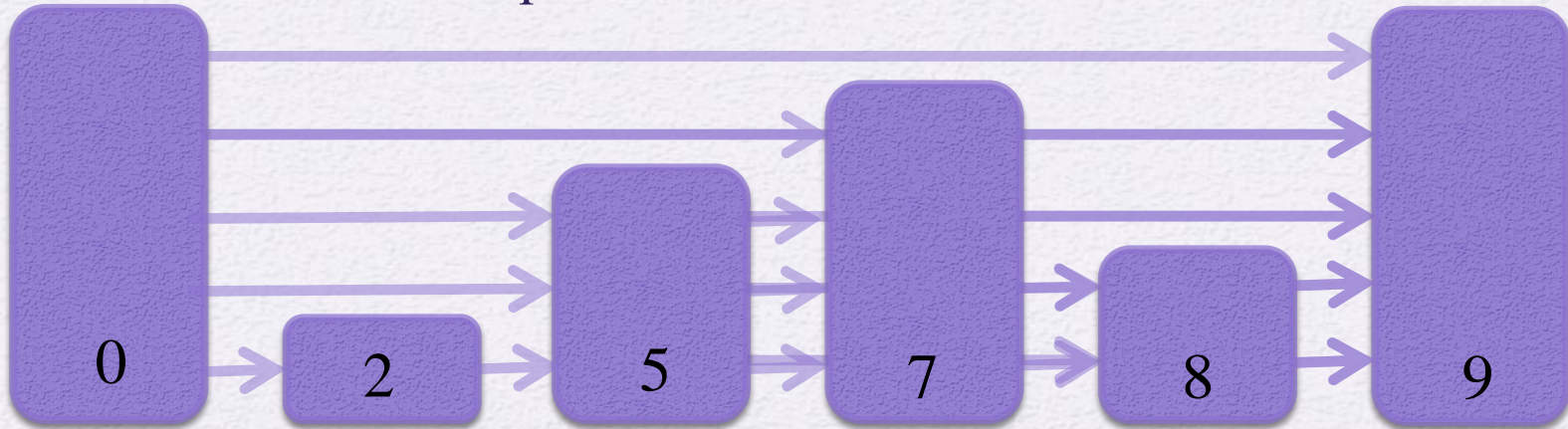
список с пропусками skip list

- При поиске мы ищем сверху вниз (пример - поиск 8)
 - Отбрасываем то, что сзади



список с пропусками skip list

- Добавление информации (пример – добавка 7)
 - Выбираем произвольной высоты элемент
 - Удаляем все транзитные связи и добавляем новые



список с пропусками skip list

- Каждый уровень номер i перепрыгивает примерно 2^i узлов
 - среднее расстояние между узлами в уровне
- Высоты узлов выбираются случайно, так что гарантии только вероятностные
- Неблокирующая реализация использует неблокирующий односвязный список для каждой «башни»
- В процессе работы каждая операция «помогает» каждой
- Детали реализации можно найти в
M. Fomitchev. Lock-free linked lists and skip lists. Master's thesis, York University, October 2003. <http://www.cs.yorku.ca/~mikhail>.

Хэш таблицы

- О таблице с открытой адресацией, реализующий конечный автомат, будет рассказано в следующей лекции
- Сейчас коротко рассмотрим неблокирующую таблицу

Неблокирующий хэш

- Предложено в
H. Gao, J. F. Groote, W. H. Hesselink, "Almost Wait-Free Resizable Hashtables," ipdps, vol. 1, pp.50a, 18th International Parallel and Distributed Processing Symposium (IPDPS'04)
- 126 операторов псевдокода реализации, из них
 - 42 – insert, find, delete
 - 30 – getAccess/releaseAccess
 - 46 – new Table, refresh, migrate, moveContents
 - 6 – moveElement
- 2 человеко-года построения алгоритмов машинной верификации, более 200 инвариантов доказано, примерно дюжина ошибок обнаружена в ходе этого процесса

Неблокирующий хеш

- Основной алгоритм
 - { getAccess; (find/delete/insert)*; releaseAccess; }
 - Считает количество обратившихся процессов
 - Использует открытую адресацию
 - помечает удаленные неудаляемым символом
 - при переполнении таблицы переносит содержание в новую
- Реализация не проста, много разных состояний

Таблица с открытой адресацией

- Предложено как упрощение Gao в Chris Purcell and Tim Harris. Non-blocking hash tables with open addressing. Distributed Computing, pages 108–121, 2005
- Достоинства
 - Все лежит только внутри таблицы, нет указателей
 - хорошая локальность по кэш линии (при отсутствии коллизий нужна только одна линия!)
 - нет “пометок удаления” - не надо реорганизовывать
 - Поддерживается прозрачная миграция и рост таблиц
- Недостатки
 - Может хранить только ключ без значения - нельзя использовать для словарей
 - ключи нельзя модифицировать, можно только удалить

Таблица с открытой адресацией

- Соисполняемые процессы добавления и удаления непросты, так как заранее неизвестно, в какую позицию попадет ключ
 - Недостаточно просто вставить в первую попавшуюся свободную позицию, так как соисполнение процедуры удаления может изменить позицию «первого» элемента, и может появиться потенциально два одинаковых ключа!
- Каждая позиция есть пара атомарно модифицируемых значений <состояние, ключ>
 - Состояния принимают значения
 - Empty busy inserting member
 - ключ определен для последних двух значений

Таблица с открытой адресацией

- Для решения проблемы с добавкой она разбита на 3 стадии
 - Резервируется пустая позиция и этот факт анонсируется помещением туда флага `inserting`
 - Проверяются другие потенциальные места, где может оказаться ключ
 - при соисполнении вставки в «процессе» или
 - уже добавленный до того
 - Если нашелся другой вставленный ключ, то текущая процедура завершается неуспешно, а позиция чистится
 - Если другой процесс вставки в процессе, то он помечается как `busy` и блокирует другие процессы
 - В последней стадии при помощи CAS статус меняется на `member`
- Основная реализация является свободной от блокировок

Таблица с открытой адресацией

```
bool CollisionInBucket(int h, int index)
{
    versionstate temp1(GetBucket(h,index)->vs);
    if (IsReadableKey((State)temp1.state))
        if (Hash(GetBucket(h,index)->key) == h )
            {
                versionstate temp2(GetBucket(h,index)->vs);
                if (IsReadableKey(temp2.state))
                    if (temp1.version == temp2.version)
                        return true;
            }
    return false;
}

void ConditionallyRaiseBound(int h, int index)
{
    Bounds old, nb;
    do
    { old = Bounds(bounds[h]);
      nb = Bounds(max(old.bound, (unsigned)index), false);
    } while (!CAS32(bounds+h, old, nb));
}
```

```
void ConditionallyLowerBound(int h, int index)
{ Bounds bs(bounds[h]); // Allow maximum < index
  if (bs.scanning == true)
      CAS32(bounds+h, bs, Bounds(bs.bound, false));

  if (index > 0) // If maximum > 0, set maximum < index
      while (CAS32(bounds+h, Bounds(index,false),
                   Bounds(index,true)))
          { int i = index-1; // Scanning: scan for new max
            while ((i>0) && ! CollisionInBucket(h,i))
                i--;
            CAS32(bounds+h, Bounds(index,true),
                  Bounds(i,false));
          }
}

bool Find(Key key)
{ Key h(Hash( key ));
  unsigned int maximum(GetProbeBound( h ));
  for( unsigned int i =0; i <= maximum; i++ )
      { versionstate vs(GetBucket(h, i)->vs); // atomic read
        if ((vs.state==Member) && (GetBucket(h,i)->key==key))
            if (GetBucket(h,i)->vs==versionstate(vs.version,Member))
                return true;
        }
  return false;
}
```


Таблица с открытой адресацией

```
bool NB_HashTable::Add(NB_Key key)
{ Key h(Hash( key )); unsigned int version;
retry:
    int i=-1; // Reserve a cell
    do
    { if (++i >= (signed)size)
        goto retry; // table full - can be only temporary
      versionstate vs=GetBucket(h,i)->vs; // atomic read
      version = vs.version;
    } while (!CAS32(&GetBucket(h,i)->vs),
              versionstate(version,Empty), versionstate(version,Busy)));
    GetBucket(h,i)->key = key;
    for(;;) // Attempt to insert a unique copy of k
    { GetBucket(h,i)->vs = versionstate(version,Visible);
      ConditionallyRaiseBound(h,i);
      GetBucket(h,i)->vs=versionstate(version,Inserting);
      bool res = Help(key,h,i,version);
      if(! (GetBucket(h,i)->vs==versionstate(version,Collided)))
          return _true;
      if (!res)
      { ConditionallyLowerBound(h,i);
        GetBucket(h,i)->vs = versionstate(version+1,Empty);
        return _false;
      }
      version++;
    }
}
```

```
bool Del(Key key) // Remove k from the set if it is a member
{
    Key h(Hash( key ));
    unsigned int maximum(GetProbeBound( h ));
    for( unsigned int i =0; i <= maximum; i++ )
    {
        versionstate vs(GetBucket(h, i)->vs); // atomic read
        if ((vs.state==Member) && (GetBucket(h,i)->key==key) )
        {
            if (CAS32(&GetBucket(h,i)->vs),
                  versionstate(vs.version,Member),
                  versionstate(vs.version,Busy)))
            {
                ConditionallyLowerBound(h,i);
                GetBucket(h,i)->vs = // atomic
                    versionstate(vs.version+1,Empty);
                return _true;
            }
        }
    }
    return false;
}
```

Таблица с открытой адресацией

```
// Return true if no other cell seen in member state
bool Help(Key key, int h, int i, int ver_i) // Attempt to
insert k at i
{
    int maximum(GetProbeBound( h )); // Scan through probe
sequence
    for( int j=0; j <= maximum; j++ )
    {
        if (i != j)
        {
            versionstate vs_j(GetBucket(h,j)->vs);
            unsigned int ver_j = vs_j.version;
            if ((vs_j.state==Inserting)&&(GetBucket(h,j)->key==key))
            {
                if (j < i) // Help inserts found in the probe sequence
                {
                    if (GetBucket(h,j)->vs==versionstate(ver_j,Inserting))
                    {
                        CAS32(&(GetBucket(h,i)->vs),
                            versionstate(ver_i,Inserting),
                            versionstate(ver_i,Collided));
                        return Help(key,h,j,ver_j);
                    }
                } else
            }
        }
    }
}
```

```
{ // Fail insert found later in the probe sequence
    if (GetBucket(h,i)->vs==versionstate(ver_i,Inserting))
    {
        CAS32(&(GetBucket(h,j)->vs),
            versionstate(ver_j,Inserting),
            versionstate(ver_j,Collided));
    }
}
}
}
vs_j = GetBucket(h,j)->vs; // Abort if k already a member
ver_j = vs_j.version;
if ( (vs_j.state == Member) && (GetBucket(h,j)->key==key) )
{
    if (GetBucket(h,j)->vs==versionstate(ver_j,Member))
    {
        CAS32(&(GetBucket(h,i)->vs), versionstate(ver_i,Inserting),
            versionstate(ver_i,Collided));
        return _false;
    }
}
}
CAS32(&(GetBucket(h,i)->vs), versionstate(ver_i,Inserting),
    versionstate(ver_i,Member));
return _true;
}
```

Выводы

- Рассмотрены несколько практических и теоретических типовых структур данных и алгоритмов работы с ними, обладающих теми или иными свойствами свободы при параллельном исполнении
- Разные направления использования
 - Счетчики, Стеки и очереди, Операции с откатами (RCU, seqlock), Список с пропусками, Хеш таблицы
- На их примере рассмотрены некоторые подходы, используемые при создании таких структур
- Увы, пока они не часто применяются на практике, сложно доказать безошибочность, а без него почти всегда есть ошибки
- Многие используют «расслабление» требований – «почти свободная от ожидания» означает для большинства случаев, кроме нечасто встречающихся на практике.
- Надо оценивать не только сложность, но и количество операций блокировки шины

(с) А. Тормасов, 2010-11 г.

Базовая кафедра «Теоретическая и Прикладная Информатика» ФУПМ МФТИ
tor@ crec .mipt .ru_

Для коммерческого использования курса просьба связаться с автором.