Синхронизация

Традиционный подход - mutex etc
Недостатки :

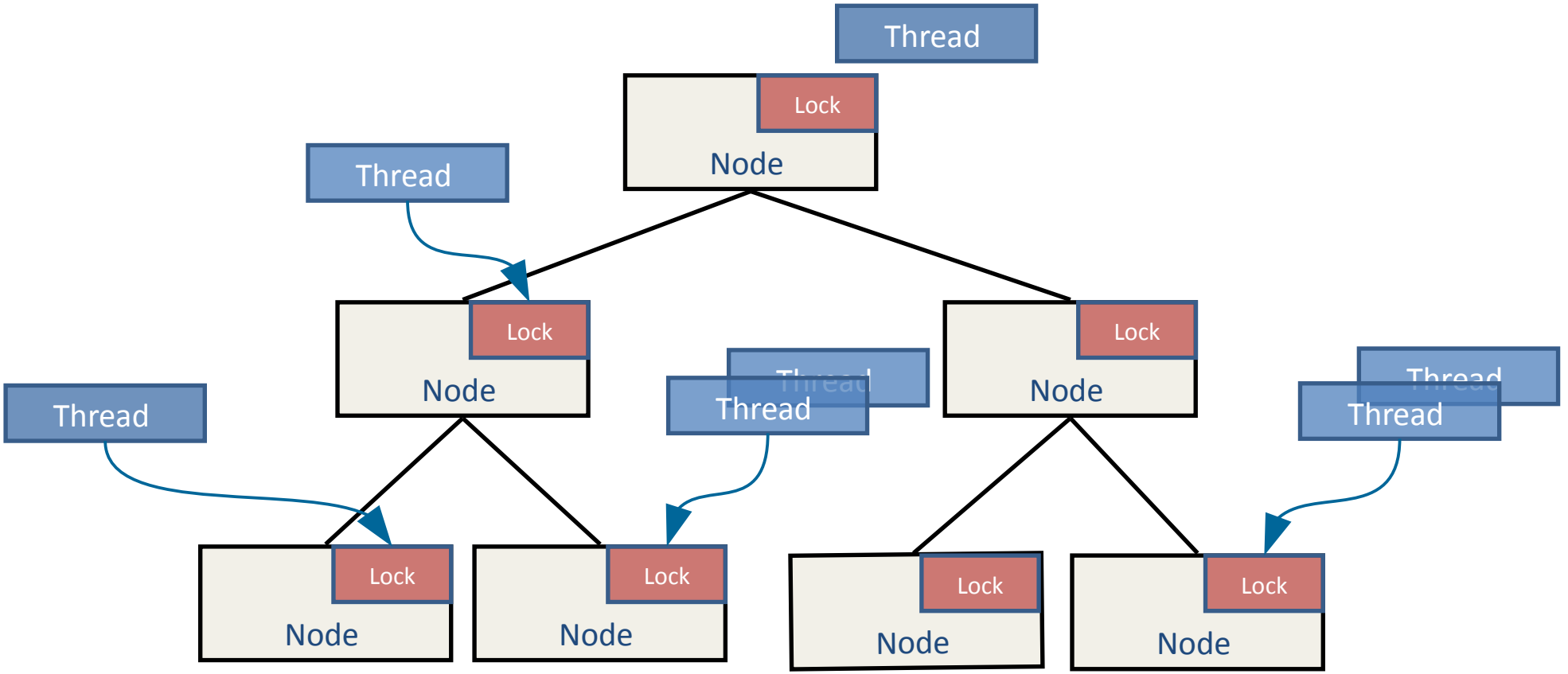
- ✗ Serialization
- ✗ Deadlock
- ✗ Priority inversion

Альтернативы

Альтернативы:

- ✔ Fine-grained locks
- ✔ Lock-free / wait-free
- ✔ Transactional memory

Fine-grained locks



Блокировка на уровне узла структуры

Применимы spinlocks

Transactional memory

```
for (;;) {
  tr_begin();
  Insert/delete node
  if ( tr_commit() )
     break;
}
```

- ✔ STM - software transaction memory

  Недостаточная производительность

- ✔ HTM - hardware transaction memory

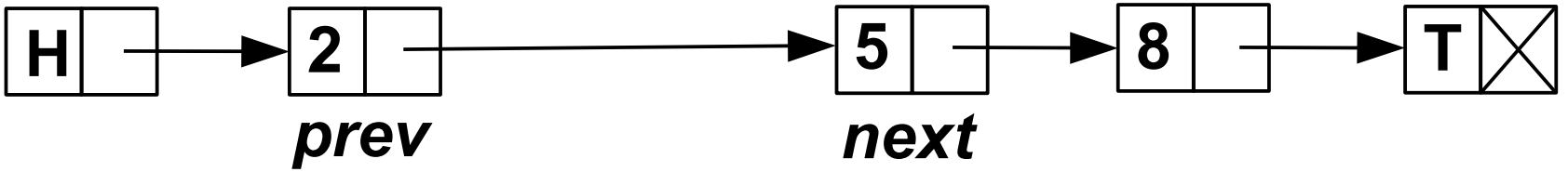
  Только появляется (Haswell, PowerPC)

CAS — compare-and-swap

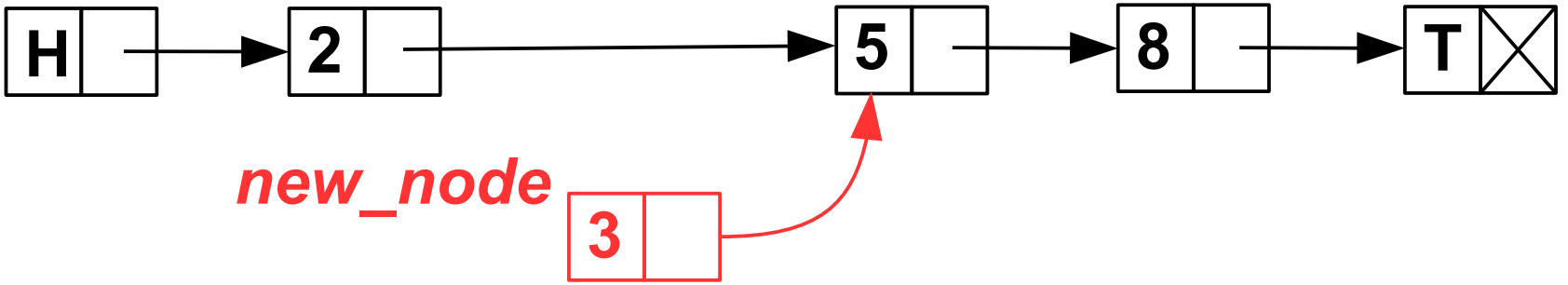
```
template <typename T>
bool CAS( T * pAtomic, T expected, T desired )
atomically {
   if ( *pAtomic == expected ) {
      *pAtomic = desired;
      return true;
   }
  else
      return false;
};
```

Lock-free list: insert

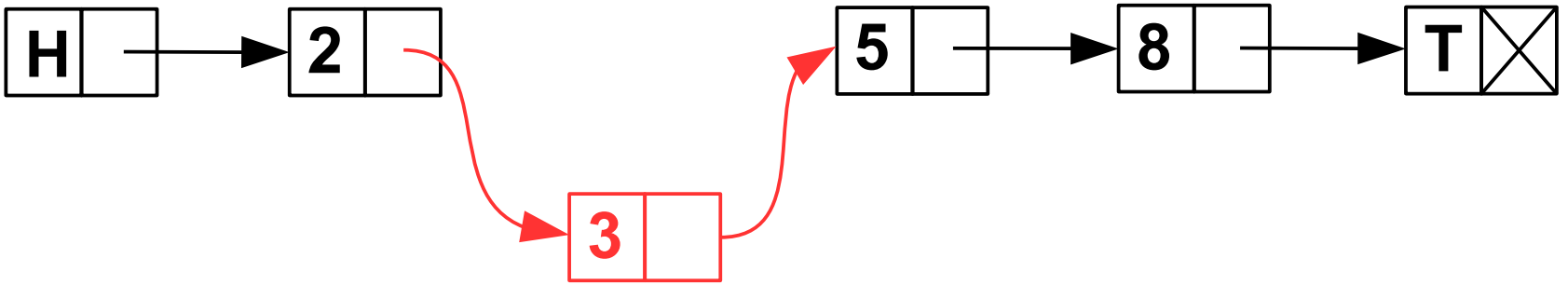
1. find insert position for key 3



2. new_node.next_.store(next)

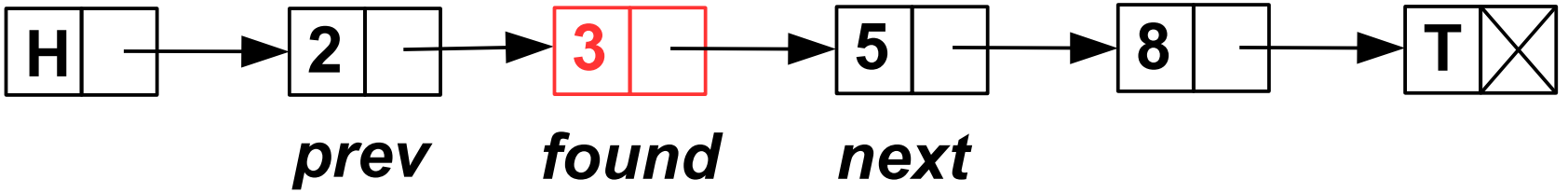


3. prev->next_.CAS(next, new_node)

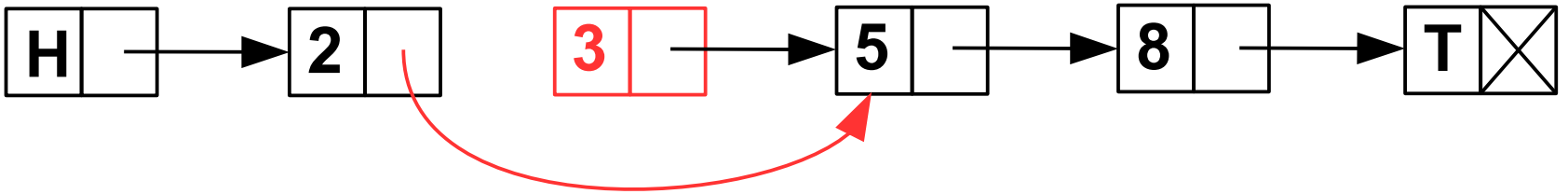


Lock-free list: erase

1. find key 3



2. prev->next_.CAS(found, next)

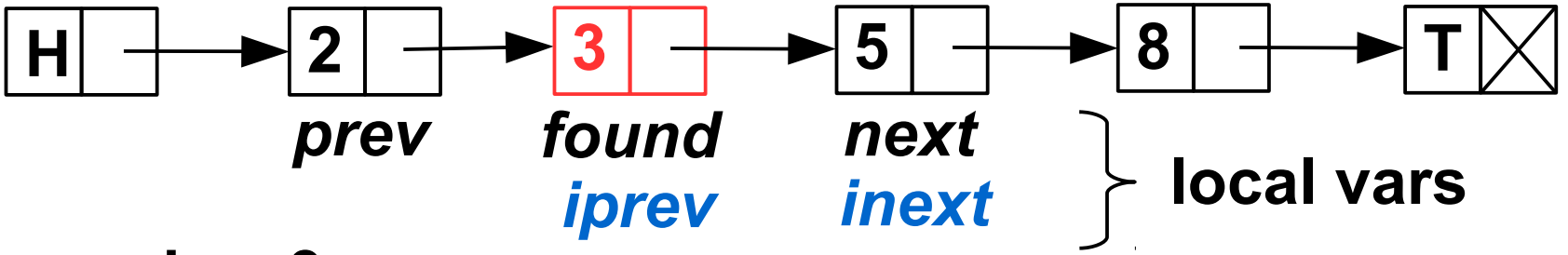


Проблема: параллельный insert

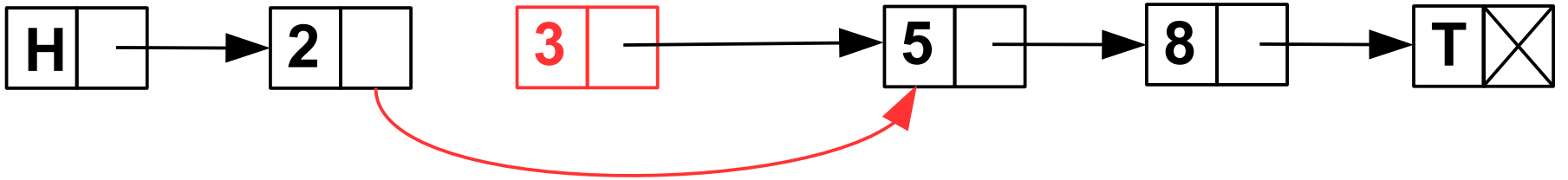
Lock-free list: insert/erase

A: find key 3

B: find insert pos for key 4

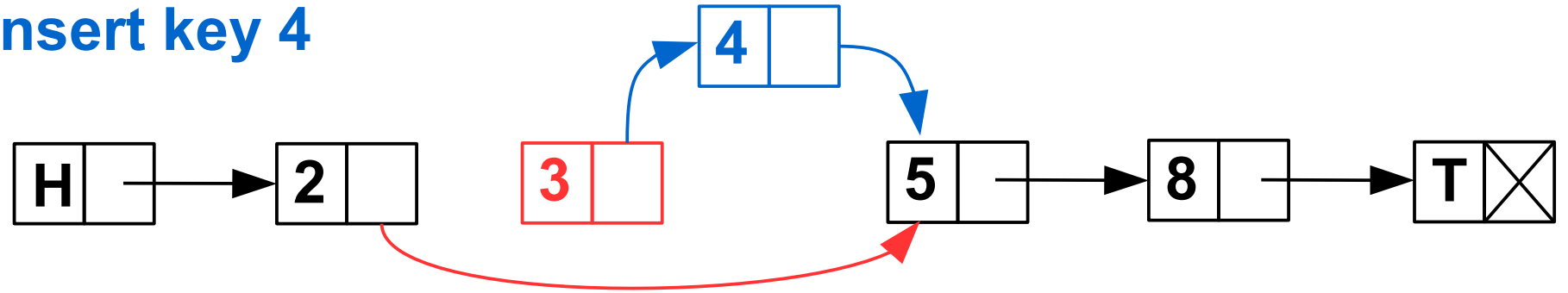


A: erase key 3



prev->next_.CAS(found, next)

B: insert key 4



iprev->next_.CAS(inext, new_item)

Marked pointer

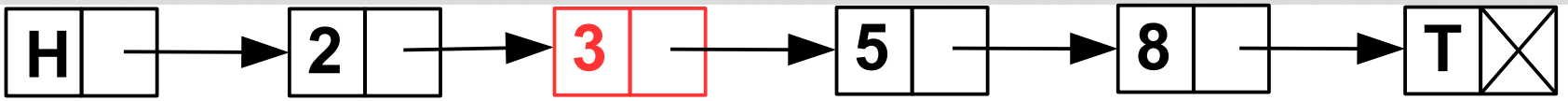
[T.Harris, 2001]

Двухфазное удаление:

- ***Логическое*** **удаление — помечаем элемент**
- ***Физическое*** **удаление — исключаем элемент**

В качестве метки используем младший бит указателя

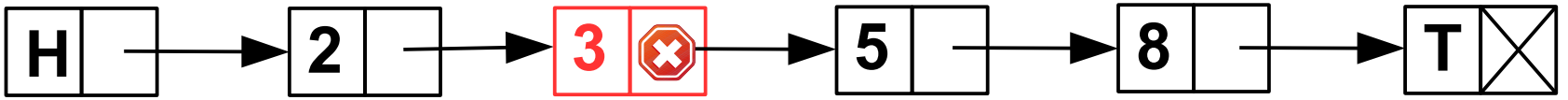
Lock-free list: marked pointer



A: erase *prev* *found* *next*

B: insert *iprev* *inext*

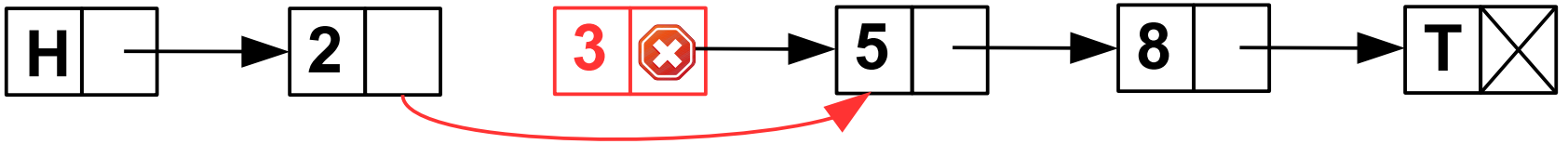
A: Logical deletion - mark item *found*



found->next_.CAS(next, next | 1)

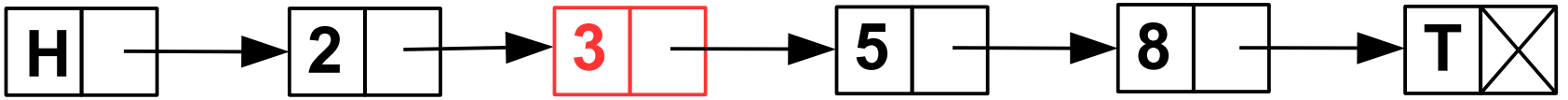
B: iprev->next_.CAS(inext, new_item) - failed!!!

A: Physical deletion - remove item *found*



prev->next_.CAS(found, next)

Lock-free list: problems



A: erase
B: insert

prev
found
iprev
inext

local vars

iprev->next_.CAS(inext, new_item)

Вдруг уже удалены?..

prev->next_.CAS(found, next)

Lock-free list: problems

Проблемы:

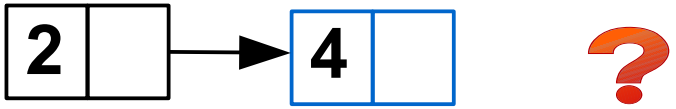
- **Защита локальных данных — когда элемент можно безопасно удалить?**
- **ABA-проблема**

ABA-проблема

Thread A: erase(3)

prev *found* *next*

preempted...

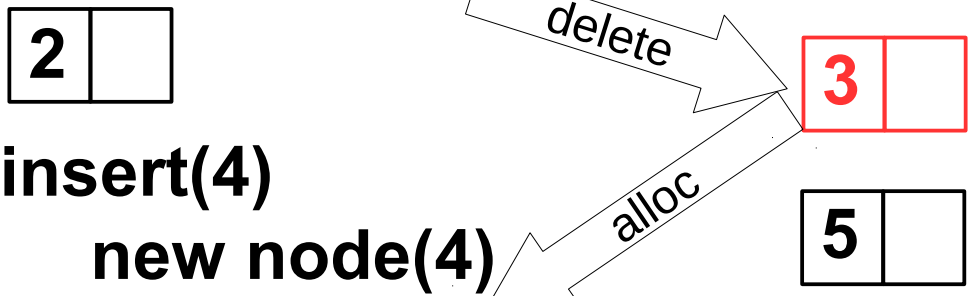


prev *found* *next*

Thread B

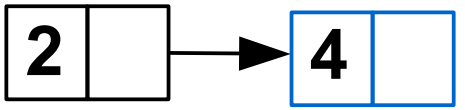
erase(3); erase(5)

Heap



insert(4)

new node(4)



addr(3) == addr(4)

prev->next_.CAS(found, next) - success!!!

мусор

SMR

Проблемы:

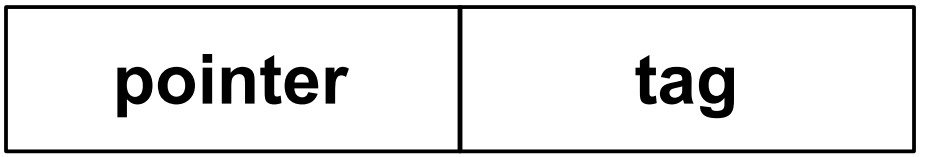
- **Защита локальных данных — когда элемент можно безопасно удалить?**
- **ABA-проблема**

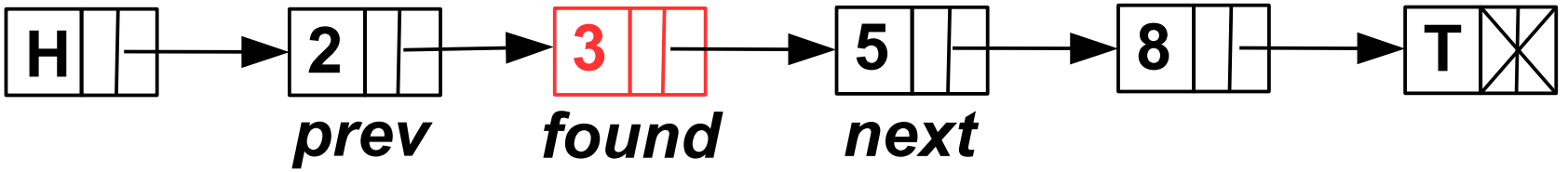
Решение:
Safe memory reclamation (SMR)

- **Tagged pointers**
- **Hazard Pointers**
- **User-space RCU**

Tagged pointers

```
template <class T>
struct tagged_ptr {
  T * ptr;
  uintptr_t  tag;
};
```





prev->next_.dwCAS(found, <next.ptr, prev->next_.tag + 1>)

- **Требует dwCAS — *не везде есть***
- **Решает только ABA-проблему**
  - *Освободить память **нельзя**, нужен free-list*

[boost.lock-free]

Tagged pointers: history

ABA-проблема характерна только для CAS

Архитектуры процессоров

LL/SC:
- **IBM PowerPC**
- **MIPS**
- **ARM**

CAS:
- **x86, amd64**
- **Sparc**
- **Itanium**

```
bool weak_CAS( T * ptr,
    T  expected, T  desired )
{   T  cur = LL( ptr );
     return cur == expected
         && SC( ptr, desired );
}
```

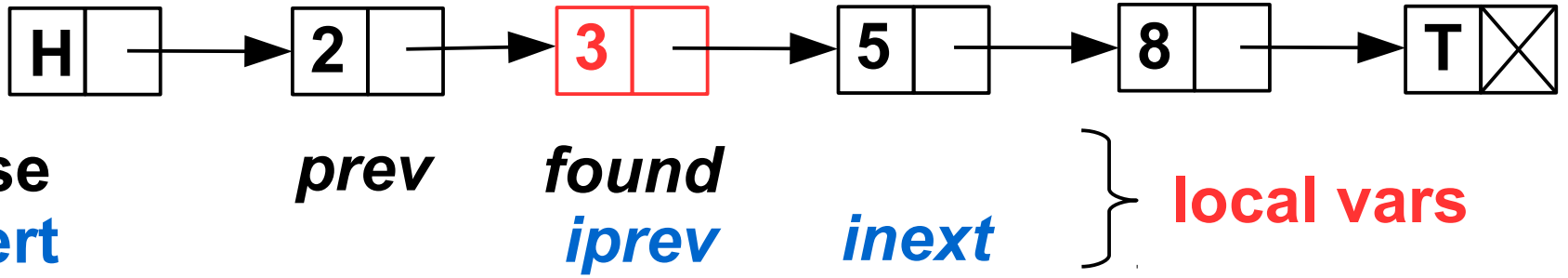
- *LL — load linked*
- *SC — store conditional*

Эмуляция LL/SC на CAS — намного труднее

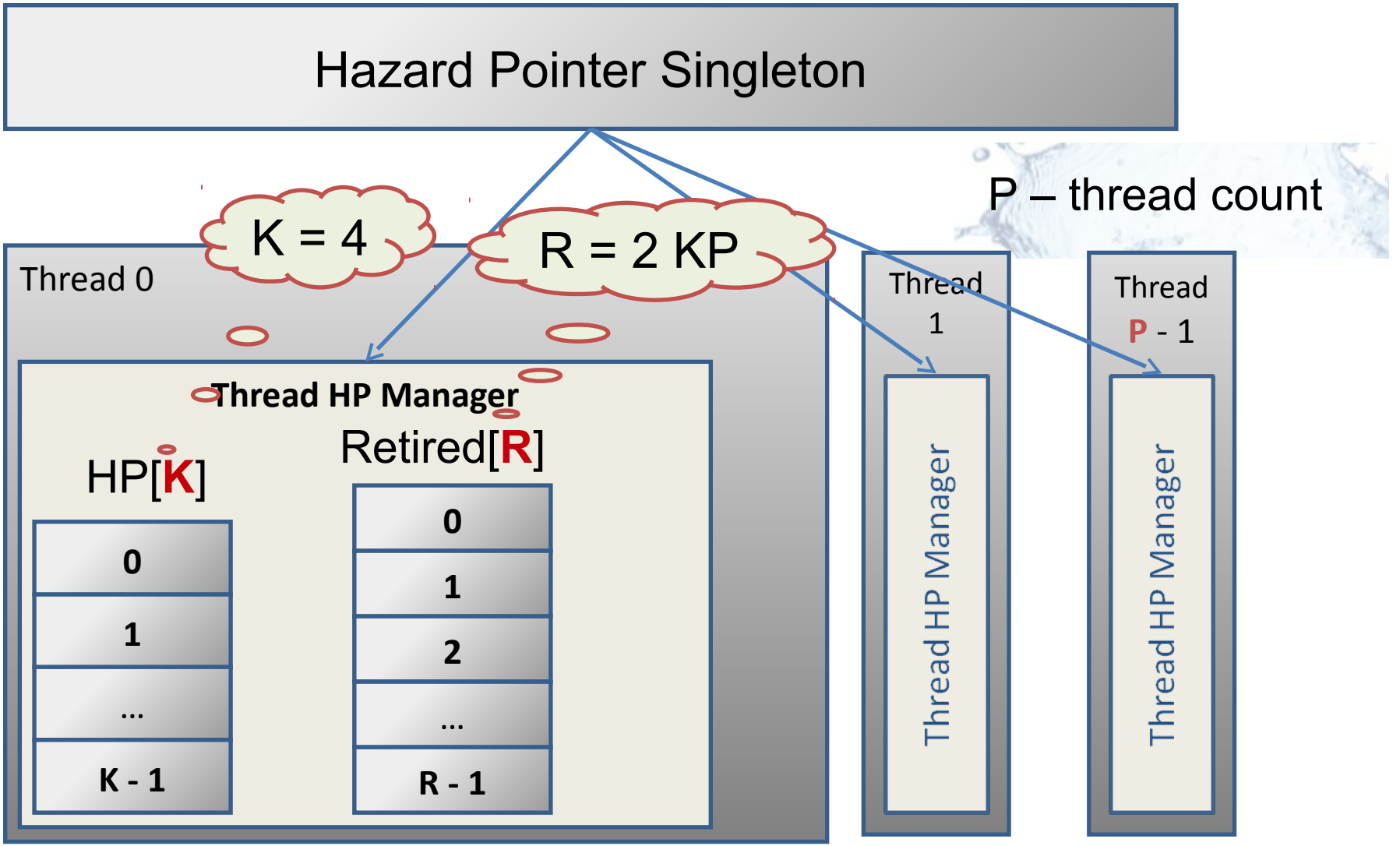
C++11 — только CAS

Hazard pointers

- ✔ решает ABA-проблему
- ✔ Физическое удаление элементов
  - ✔ Использует только атомарные чтение/запись
  - ◆ Защищает только *локальные* ссылки
  - ✔ Размер массива отложенных (готовых к удалению) элементов *ограничен сверху*
  - ◆ Перед работой с указателем его следует объявить как *hazard*

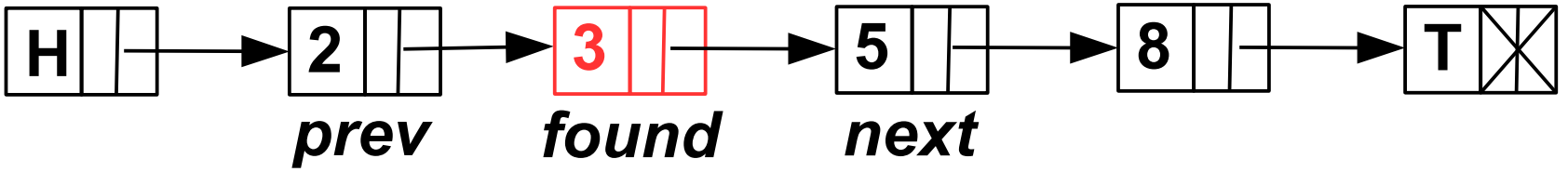


Hazard pointers



<K,P, R> : R > K * P

Hazard pointers




```
erase( Key k ) {
    hp_guard h1 = get_guard();
    hp_guard h2 = get_guard();
retry:
    node * prev = Head;
    do {
        node * found = h2.protect( prev->next_);
        if ( found->key == k )
            if (prev->next_.CAS( found, found->next_)) {
                hp_retire( found );
                return true;
            }
            else
              goto retry;
        h1 = h2;
        prev = found;
    } while ( found->key < k );
    return false;
}
```

Распределяем HP (TLS)

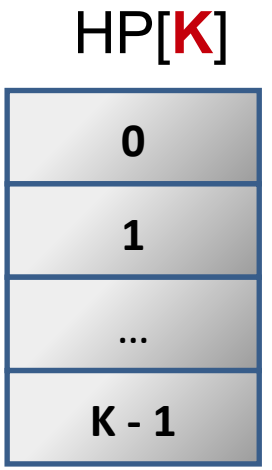
Защищаем элемент

Удаляем элемент

Hazard Pointers

Объявление Hazard Pointer'a – защита локальной ссылки

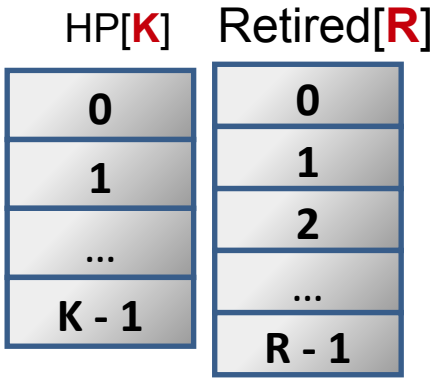
```
class hp_guard {
   void * hp;
   // ...
};
T * hp_guard::protect(
      std::atomic<T*>& what) {
  T * t;
  do {
    hp = t = what.load();
  } while (t != what.load());
  return t;
}
```



Hazard Pointers

Удаление элемента

```
void hp_retire( T * what ) {
    push what to current_thread.Retired array
    if ( current_thread.Retired is full )
        hp.Scan( current_thread );
}

void hp::Scan() {
   void * guarded[K*P] = union HP[K] for all P thread;
   foreach ( p in current_thread.Retired[R] )
      if ( p not in guarded[] )
         delete p;
}
```



Гарантия scan(): **<K,P, R> : R > K * P**

User-space Read-Copy Update

- ✔ решает ABA-проблему
- ✔ Физическое удаление элементов

- ✔ RCU — метод *синхронизации:*
  RCU.lock() / RCU.unlock()
- ✔ Разработан для *почти*-read-only данных (map, set)
- ✔ Очень легкие read-side lock
- ✔ Удаление элемента — ожидание окончания эпохи:
  RCU.sync()

RCU.lock(): ничего не блокирует
Объявляет, что поток входит в текущую RCU-эпоху

User-space RCU

Thread 1

```
Map.find( ... );
Set.insert( ... );
Map.find( ... );
Map.erase( ... )...
```

Thread N

```
Set.find( ... );
Map.insert( ... );
Set.find( ... );
Set.insert( ... );
```

Эпоха 1

++Эпоха

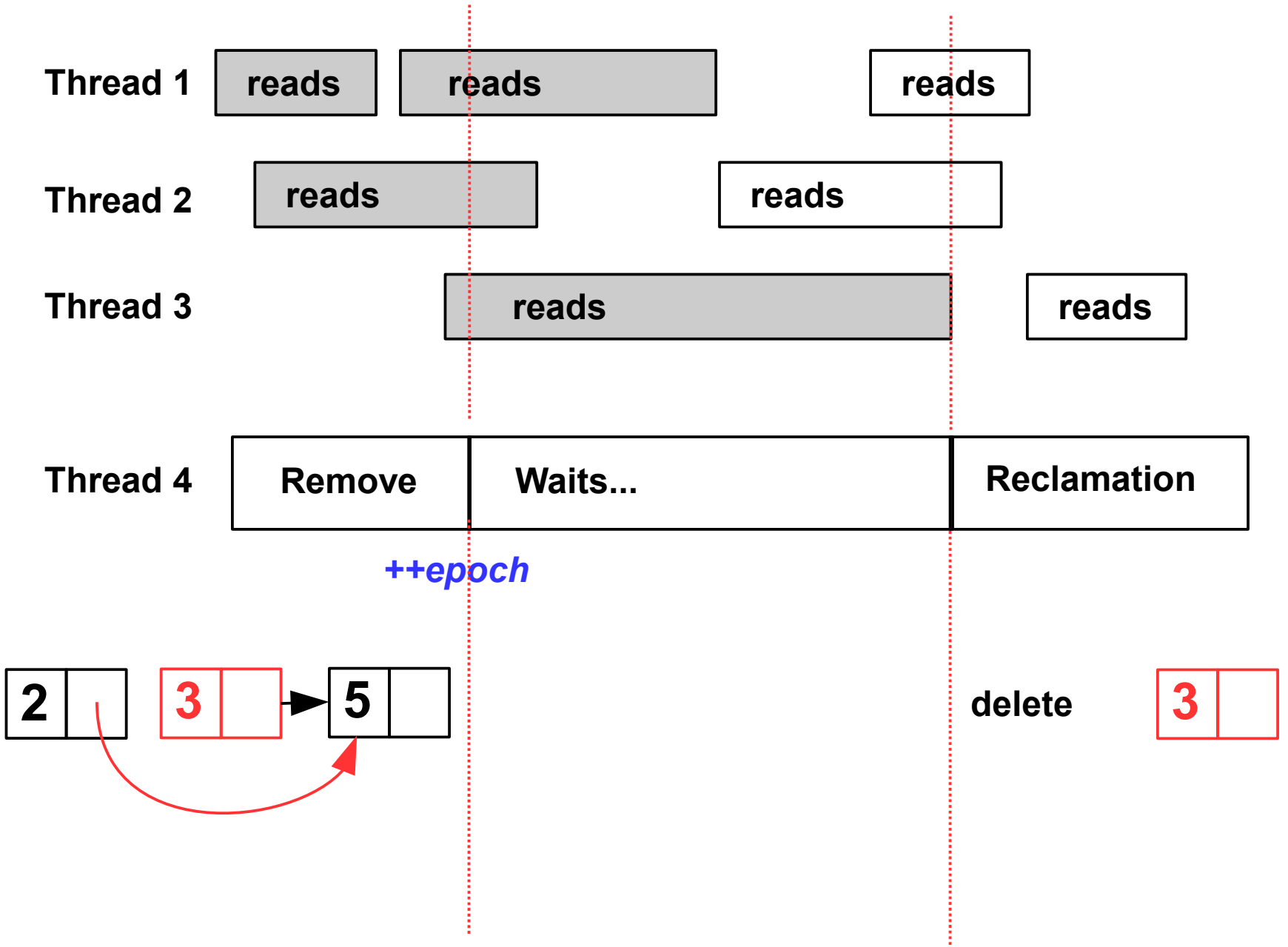
RCU.sync() - ждем, пока все потоки покинут эпоху 1

Эпоха 2

```
... Map.erase;
Map.find( ... );
Set.insert( ... );
Map.find( ... );
```

```
Set.find( ... );
Map.insert( ... );
Set.find( ... );
Set.insert( ... );
```

User-space RCU



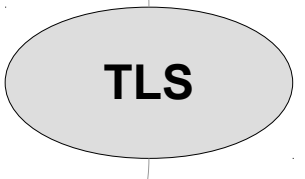
User-space RCU

```
//Global data
std::atomic<uint32_t> globalCtl(1);
std::mutex            rcu_mutex;

// Per-thread data
struct rcu_thread_data {
  std::atomic<uint32_t> ctl;
  rcu_thread_data * next;

  rcu_thread_data(): ctl(0), next(nullptr) {}
};
```


```
const uint32_t c_nNestMask = 0x7FFFFFFF
```

User-space RCU

```
void rcu_enter() //вход в RCU
{
    rcu_thread_data * rec = get_thread_record();
    uint32_t tmp = rec->ctl.load();
    If ( (tmp & c_nNestMask) == 0 )
        rec->ctl.store( globalCtl.load());
    else // inc nested counter
        rec->ctl.fetch_add( 1 );
}

void rcu_leave() //выход из RCU
{
    rcu_thread_data * rec = get_thread_record();
    rec->ctl.fetch_sub( 1 );
}
```



User-space RCU

```
void rcu_sync() // переход в новую эпоху
{
    std::unique_lock<std::mutex> sl(rcu_mutex);
    flip_and_wait();
    flip_and_wait();
}

void flip_and_wait() {
    // start new epoch
    globalCtl.fetch_xor( 0x80000000 );
    // ждем пока все потоки не выйдут из эпохи
    foreach ( rcu_thread_data* rec ) {
      while ( !( rec в эпохе globalCtl ))
          yield(); // back-off
    }
}
```

User-space RCU

Thread A `globalCtl.epoch = 0` **Thread B**

```
rcu_enter()
    tmp=globalCtl.load()
    // epoch=0

    rec→ctl = tmp;
    // thread epoch=0

    ...
    // traverse data

rcu_leave()
```

```
rcu_sync()
    flip_and_wait();
    // globalCtl.epoch=1

rcu_sync()
    flip_and_wait();
    // globalCtl.epoch=0

delete node;
```