Теория и практика многопоточного программирования

Неганов Алексей

Московский физико-технический институт (национальный исследовательский университет)
Кафедра теоретической и прикладной информатики

Москва 2020

Вспомним теорию

Виды синхронизации:

- Coarse-grained
- Fine-grained
- Optimistic
- Lazy
- Nonblocking

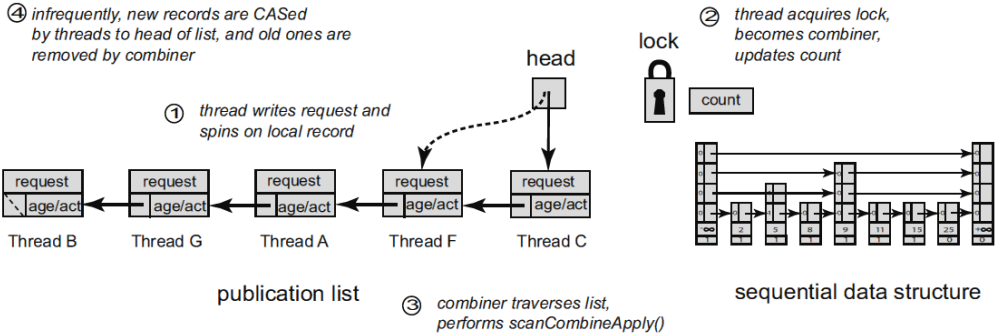
Корректность алгоритма:

- Инварианты
- Линеаризуемость

Flat combining

```
template <class T> class LockStack {
    std::stack<T *> m_Stack;
    std::mutex      m_Mutex;
public:
    void push( T& v ) {
        m_Mutex.lock();
        m_Stack.push( &v );
        m_Mutex.unlock();
    }
    T * pop() {
        m_Mutex.lock();
        T * pv = m_Stack.top();
        m_Stack.pop()
        m_Mutex.unlock();
        return pv;
    }
};
```

Flat combining



https://www.cs.bgu.ac.il/~hendlerd/papers/flat-combining.pdf

Lock-free stack

```
template<typename T> class lock_free_stack {
struct node {
    T data;
    node* next;
    node(T const& data_): data(data_) {}
};
std::atomic<node*> head;
public:
    void push(T const& data) {
        node* const new_node=new node(data);
        new_node->next=head.load();
        while(!head.compare_exchange_weak(new_node->next,new_node));
    }
};
```

Lock-free stack: ABA

```
void simple_pop(T& result) {
    node *old_head=head.load();
    while(!head.compare_exchange_weak(old_head, old_head->next));
    result=old_head->data;
}
```

Lock-free stack: free list

```
template<typename T> class lock_free_stack {
    std::atomic<unsigned> threads_in_pop;
    std::atomic<node*> to_be_deleted;
    void try_reclaim(node* old_head);
public:
    std::shared_ptr<T> pop() {
        ++threads_in_pop;
        node* old_head=head.load();
        while(old_head && !head.compare_exchange_weak(old_head, old_head->next));
        std::shared_ptr<T> res;
        if(old_head)
            res.swap(old_head->data);
        try_reclaim(old_head);
        return res;
    }
};
```

Lock-free stack: hazard pointers

```
std::shared_ptr<T> pop() {
    std::atomic<void*>& hp=get_hazard_pointer_for_current_thread();
    node *old_head=head.load();
    do {
        node *temp;
        do {
            temp = old_head;
            hp.store(old_head);
            old_head=head.load();
        } while(old_head!=temp);
    } while(old_head && !head.compare_exchange_strong(old_head,old_head->next));
    hp.store(nullptr);
    std::shared_ptr<T> res;
    if(old_head) {
        res.swap(old_head->data);
        if(outstanding_hazard_pointers_for(old_head))
            reclaim_later(old_head);
        else
            delete old_head;
        delete_nodes_with_no_hazards();
    }
    return res;
}
```

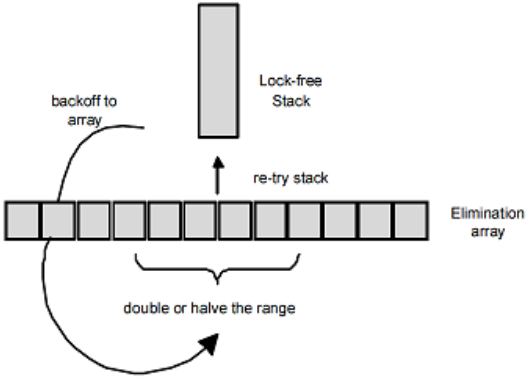
Lock-free stack: backoff

```
void push(T const& data) {
    node *const new_node=new node(data);
    node *t = head.load(std::memory_order_relaxed);
    while (1) {
        new_node->next.store(t, std::memory_order_relaxed);
        if (head.compare_exchange_weak(t, new_node, std::memory_order_release, std::memory_order_relaxed))
           return;
        bkoff();
    }
}

node *pop() {
   typename gc::Guard guard; // Hazard pointer guard
   while (1) {
      node *t = guard.protect(head);
      if ( t == nullptr )
         return nullptr ;  // stack is empty

      node *pNext = t->next.load(std::memory_order_relaxed);
      if ( head.compare_exchange_weak(t, pNext, std::memory_order_acquire, std::memory_order_relaxed ))
           return t;
      bkoff();
   }
}
```

Elimination backoff



Elimination backoff