# Практическое занятие Программирование без блокировок

## Задание 1. Реализация spinlock.

1. Создайте класс таким образом, чтобы его можно было использовать в качестве мьютекса, но без блокировки потока. Реализуйте методы lock(), unlock() и try\_lock()  
   При реализации требуется предусмотреть максимально возможную производительность => подумать над заданием соответствующего упорядочения
2. Подсказка – разработанный класс можно использовать в обертках типа lock\_guard или unique\_guard
3. Для проверки работоспособности замените мьютекс в реализации потокобезопасного стека (или очереди) на разработанный класс
4. Оцените (качественно) полученный выигрыш на одном и том же наборе данных при использовании обычного мьютекса и разработанного spinlock-а

## Задание 2. Реализация lock-free стека на основе «сборщика мусора»

Требуется разработать класс lock\_free\_stack без использования блокировок. Позволяем нескольким потокам добавлять данные (push) в стек, и нескольким потокам извлекать данные из стека для обработки

1. Простейшая реализация - посредством односвязного списка
2. Класс должен предоставлять пользователю:  
   конструктор  
   деструктор  
   добавление нового узла – push()  
   удаление узла «с верхушки стека» - pop()  
   все остальные операции запрещаем!

### Вспомогательный класс Node

Данные:

* Несмотря на затраты, для того, чтобы безопасно извлекать данные из стека, Node содержит не встроенное данное   
  T data;  
  , а   
  shared\_ptr<T> data;
* указатель на следующий Node

Методы:

* конструктор Node(const T&)

### Класс lock\_free\_stack

Данные:

std::atomic<Node\*> head;

### Добавление узла в однопоточной программе – push(T):

В многопоточной программе – гонка между 2 и 3!!! => чтобы избежать гонки, операция «вставки» нового узла в список должна быть атомарной =>

Создать новый узел с указанным данным

Записать в его next текущее значение head

Записать в head указатель на новый узел

1

2

3

1. Node\* head – должен быть атомарной переменной для того, чтобы с ней можно было использовать атомарные операции
2. при реализации push():
   1. создать и максимально подготовить к использованию новый динамический Node (сформировать в нем данное и next)
   2. с помощью compare\_exchange\_strong() или в данном случае эффективнее с помощью compare\_exchange\_weak() обеспечиваем «вставку» нового узла в начало списка:   
      ждем пока у нового узла next не совпадает с head (это означает, что другой поток успел вставить свой узел). При совпадении новый узел вставлен в список!

### Удаление узла с верхушки стека – pop()

В однопоточной программе:

Прочитать текущее значение head

Прочитать head->next

Записать в head значение head->next

1

2

3

Сформировать возвращаемое значение

4

Удалить узел

5

В многопоточной программе эта операция гораздо сложнее, чем вставка, так как гонка может возникать как при получении одного и того же значения head в разных потоках (решается аналогично push), так и при удалении узла из списка. Также требуется учесть, что во время удаления другой поток может как добавлять новый узел в список, так и извлекать.

Проблема – удалить узел можно при условии, что никакой другой поток не вызывает pop() => нужно подсчитывать потоки, которые вызывают pop() =>  
std::atomic<unsigned int> threads\_in\_pop;   
счетчик, который увеличивается при входе, а уменьшать при выходе.

Также нужно формировать отдельный список узлов, подлежащих удалению =>

std::atomic<Node\*> to\_be\_deleted;

Псевдокод pop()

std::shared\_ptr<T> pop()

{

++ threads\_in\_pop;

//получаем указатель на текущий head

Node\* old\_head = …

//пока old\_head не нулевой (стек пустой) && head перенаправлен на следующий Node (compare\_exchange\_weak) при условии, что на момент обмена head не изменился

while(…);

//формируем возвращаемое значение:  
 std::shared\_ptr<T> res; //данное по умолчанию  
 if(old\_head){ res.swap(old\_head->data);} //не копируем указатель, а обмениваем данные

//Пробуем удалить ненужные узлы

try\_reclaim(old\_head); //вызываем вспомогательный метод

return res;

}

Вспомогательный метод:

void try\_reclaim(Node\* old\_head)

{

if(threads\_in\_pop == 1) //единственный в pop() => попытка удалить

{

//можно безопасно удалять только что исключенный из списка old\_head  
delete old\_head;

//Пытаемся удалить накопившиеся исключенные узлы  
Node\* nodes\_to\_delete = to\_be\_deleted.exchange(nullptr); //захватить список на удаление

if(!--threads\_in\_pop)//точно единственный?

{

//удаляем узлы, накопившиеся в списке nodes\_to\_delete

while(nodes\_to\_delete){ … }

}else if( nodes\_to\_delete) {chain\_pending\_nodes(nodes\_to\_delete);} //если в захваченном списке что-то было, вернуть это в общий список узлов на удаление

}else {//удалять old\_head сейчас нельзя => добавим в список для удаления (удалим как-нибудь потом)

chain\_pending\_node(old\_head);

--threads\_in\_pop;

}

}

}

//добавляем захваченный список в общий список узлов, подлежащих удалению

void chain\_pending\_nodes(Node\* nodes)

{

Node\* last = nodes;

while(Node\* next = last->next) {last = next;}//доходим до конца списка

chain\_pending\_nodes(nodes, last);

}

void chain\_pending\_nodes(Node\* first, Node\* last)

{

last->next = to\_be\_deleted;

while(!to\_be\_deleted.compare\_exchange\_weak(last->next, first));

}

void chain\_pending\_node(Node\* n)

{

chain\_pending\_nodes(n,n);

}