Урок Спринт 2.15

Lock-Free список

Для реализации Lock-Free списка можно использовать атомарные операции для обновления ссылок между элементами списка. Вот пример кода на Go, демонстрирующий реализацию Lock-Free списка:

```
package main

import (
    "fmt"
    "sync/atomic"
    "unsafe"
)

// Node представляет узел в связанном списке.
type Node struct {
    value int           // Значение узла
    next  unsafe.Pointer // Указатель на следующий узел в списке
}

// List представляет связанный список.
type List struct {
    head unsafe.Pointer // Указатель на голову списка
}

// Add добавляет новый узел с заданным значением в начало списка.
func (l *List) Add(value int) {
    node := &Node{value: value}

    for {
        // Загружаем текущую голову списка
        oldHead := atomic.LoadPointer(&l.head)
        // Устанавливаем новый узел как следующий для добавляемого узла
        node.next = oldHead

        // Если голову можно атомарно заменить на новый узел, то прерываем цикл
        if atomic.CompareAndSwapPointer(&l.head, oldHead, unsafe.Pointer(node)) {
            break
        }
    }
}
```

```
// Print выводит значения узлов списка.
func (l *List) Print() {
    // Загружаем голову списка
    curr := atomic.LoadPointer(&l.head)

    // Проходим по всем узлам в списке
    for curr != nil {
        // Преобразуем указатель в структуру Node
        node := (*Node)(curr)
        // Выводим значение узла
        fmt.Println(node.value)
        // Загружаем указатель на следующий узел
        curr = atomic.LoadPointer(&node.next)
    }
}

func main() {
    // Создаем новый связанный список
    list := &List{}
    // Добавляем узлы со значениями 1, 2, 3 в начало списка
    list.Add(1)
    list.Add(2)
    list.Add(3)
    // Выводим значения узлов списка
    list.Print()
}
```

В этом примере мы создаём структуру данных `List`, которая представляет собой `Lock-Free` список. Метод `Add` добавляет новый элемент в список, используя атомарные операции для обновления ссылок между элементами. Метод `Print` выводит значения элементов списка. Мы используем указатель `unsafe.Pointer` для представления указателей на узлы списка без блокировок.

Теперь давайте разберём, что происходит в каждом методе еще подробнее:

1. Add(value int):

- Создаётся новый узел `node` с заданным значением.
- В цикле:
  - Загружается текущая голова списка `oldHead`.
  - Устанавливается `node.next` в `oldHead`, что делает новый узел новой головой.
  - Выполняется атомарная попытка заменить текущую голову на новый узел. Если успешно, цикл завершается, иначе повторяется.

2. Print():

- Загружается указатель на голову списка `curr`.
- В цикле:

- Преобразуется указатель `curr` в структуру `Node`.
- Выводится значение узла.
- Загружается указатель на следующий узел. Если он не равен `nil`, цикл продолжается.

```
package main

import (
    "fmt"
    "sync/atomic"
    "unsafe"
)

// Node представляет узел в стеке.
type Node struct {
    value int           // Значение узла
    next  unsafe.Pointer // Указатель на следующий узел в стеке
}

// Stack представляет lock-free стек.
type Stack struct {
    top unsafe.Pointer // Указатель на вершину стека
}

// Push добавляет новый элемент на вершину стека.
func (s *Stack) Push(value int) {
    node := &Node{value: value}

    for {
        // Загружаем текущую вершину стека
        oldTop := atomic.LoadPointer(&s.top)
        // Устанавливаем новый узел как следующий для добавляемого узла
        node.next = oldTop

        // Если вершину можно атомарно заменить на новый узел, то прерываем цикл
        if atomic.CompareAndSwapPointer(&s.top, oldTop, unsafe.Pointer(node)) {
            break
        }
    }
}

// Pop удаляет и возвращает элемент с вершины стека. Если стек пуст, возвращает false.
func (s *Stack) Pop() (int, bool) {
    for {
        // Загружаем текущую вершину стека
        oldTop := atomic.LoadPointer(&s.top)
        // Если стек пуст, возвращаем false
        if oldTop == nil {
            return 0, false
        }

        // Загружаем указатель на следующий узел
```

```
            newTop := (*Node)(oldTop).next

            // Выполняем атомарную попытку заменить текущую вершину на следующий узел
            if atomic.CompareAndSwapPointer(&s.top, oldTop, unsafe.Pointer(newTop)) {
                // Возвращаем значение удалённого узла
                return (*Node)(oldTop).value, true
            }
        }
    }

    func main() {
        // Создаём новый стек
        stack := &Stack{}

        // Добавляем элементы на вершину стека
        stack.Push(1)
        stack.Push(2)
        stack.Push(3)

        // Удаляем элементы с вершины стека и выводим их значения
        for {
            if value, ok := stack.Pop(); ok {
                fmt.Println(value)
            } else {
                break
            }
        }
    }
```

В этом примере мы создаём структуру данных `Stack`, которая представляет собой `lock-free` стек. Метод `Push` добавляет новый элемент на вершину стека, используя атомарные операции для обновления ссылок между элементами. Метод `Pop` удаляет и возвращает элемент с вершины стека.

Давайте разберём, что происходит вообще:

1. Push(value int):

- Создается новый узел `node` с заданным значением.
- В цикле:
  - Загружается текущая вершина стека `oldTop`.
  - Устанавливается `node.next` в `oldTop`, что делает новый узел новой вершиной стека.
  - Выполняется атомарная попытка заменить текущую вершину на новый узел. Если успешно, цикл завершается, иначе повторяется.

2. Pop() (int, bool):

- В цикле:
  - Загружается текущая вершина стека `oldTop`.

- Если вершина равна `nil`, значит стек пуст, и возвращается значение по умолчанию и `false`.
- Загружается указатель на следующий узел `newTop`.
- Выполняется атомарная попытка заменить текущую вершину на следующий узел. Если успешно, возвращается значение удаленного узла и `true`. Если неудача, цикл повторяется.

Справка