Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра: 806 "Вычислительная математика и программирование"
Факультет: "Информационные технологии и прикладная математика"
Дисциплина: "Объектно-ориентированное программирование"

Группа:
Студент: Пашкевич Андрей Романович
Преподаватель: Поповкин Александр Викторович

Москва, 2017

# ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №8

**Вариант №17**
**Фигуры:** треугольник, квадрат, прямоугольник
**Контейнер 1-го уровня:** бинарное дерево
**Контейнер 2-го уровня:** очередь

## ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

- Знакомство с параллельным программированием в C++.

## ЗАДАНИЕ

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класса-контейнера.

**Необходимо разработать два вида алгоритма:**

- Обычный, без параллельных вызовов.
- С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

**Для создания потоков использовать механизмы:**

- future
- packaged_task/async

**Для обеспечения потоко-безопасности структур данных использовать:**

- mutex
- lock_guard

**Нельзя использовать:**

- Стандартные контейнеры std.

**Программа должна позволять:**

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.
- Проводить сортировку контейнера

## ЛИСТИНГ ПРОГРАММЫ

В программе используются контейнер первого уровня (бинарное дерево) и классы фигур, разработанные для лабораторной работы №6.

**Tree.cpp**

```
#include "tree.h"

#include <exception>
#include <functional>
#include <sstream>
#include <iostream>
#include <iomanip>

...

// сортировка обычный вызов

template <class T> void Tree<T>::sort()
{
    if (size() > 1)
    {
        size_t a = size() / 2;
        size_t i = 0;
        std::shared_ptr<T> middle = head->GetFigure();
        for (auto it : *this)
        {
            if (a == i)
            {
                middle = it;
                break;
            }
            ++i;
        }
        //std::shared_ptr<T> middle = (std::shared_ptr<T>&)this[a];
       // так на работает

        Tree<T> _left, _right;
        for (auto iter : *this)
        {
            if ((*iter).area() < middle->area())
            {
                _left.add(iter);
            }
            if ((*iter).area() > middle->area())
            {
                _right.add(iter);
            }
        }
        _left.sort();
        _right.sort();

        this->head = std::shared_ptr<Node<T>>(new Node<T>(middle));
        this->head->SetLeft(_left.head);
        this->head->SetRight(_right.head);
    }
}

// создание отдельного потока для сортировки
template<class T > std::future<void> Tree<T>::sort_in_background()
{
    std::packaged_task<void(void)> task(std::bind(std::mem_fn(&Tree<T>::sort_parallel), this));
    std::future<void> res = task.get_future();
    std::lock_guard<std::mutex> lock(mut);
    std::thread th(std::move(task));
    th.detach();
    return res;
}
```

```
// сортировка с использованием потоков
template <class T> void Tree<T>::sort_parallel()
{
    if (size() > 1)
    {
        size_t a = size() / 2, i = 0;
        std::cout << a << std::endl;
        std::shared_ptr<T> middle = head->GetFigure();
        for (auto it : *this)
        {
            if (a == i)
            {
                middle = it;
                break;
            }
            ++i;
        }

        Tree<T> _left, _right;
        for (auto iter : *this)
        {
            if ((*iter).area() < middle->area())
            {
                _left.add(iter);
            }
            if ((*iter).area() > middle->area())
            {
                _right.add(iter);
            }
        }

        std::future<void> left_res = _left.sort_in_background();
        std::future<void> right_res =_right.sort_in_background();

        head = std::shared_ptr<Node<T>>(new Node<T>(middle));;
        left_res.get();
        head->SetLeft(_left.head);
        right_res.get();
        head->SetRight(_right.head);
    }
}

template <class T> void Tree<T>::print()
{
    std::shared_ptr<Node<T>> item = head;
    std::cout << "Tree out: " << std::endl;
    item->print(0);
}

#include "figure.h"
template class Tree<Figure>;
template std::ostream& operator<<(std::ostream& os, const Tree<Figure>& tree);

https://github.com/Andrew-Bir/MAIfaq8/tree/master/oop/LAB_08
```

## ВЫВОДЫ

Параллельное программирование служит для создания программ, эффективно использующих вычислительные ресурсы за счет одновременного исполнения кода на нескольких вычислительных узлах. Параллельное программирование является более сложным по сравнению с последовательным как в написании кода, так и в его отладке.

Для обеспечения процесса параллельного программирования в C++ существуют специализированные классы.

Параллельный алгоритм, противопоставляемый традиционным последовательным алгоритмам, — алгоритм, который может быть реализован по частям на множестве различных вычислительных устройств с последующим объединением полученных результатов и получением корректного результата.

Пото́к выполне́ния (тред; от англ. thread — нить) — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы.

Мьютексы — это один из вариантов семафорных механизмов для организации взаимного исключения. Они реализованы во многих ОС, их основное назначение — организация взаимного исключения для потоков из одного и того же или из разных процессов.

Шаблонный класс std::future обеспечивает механизм доступа к результатам асинхронных операций:

- Асинхронные операции (созданные с помощью std::async, std::packaged_task) могут вернуть объект типа std::future создателю этой операции.
- Создатель асинхронной операции может использовать различные методы запроса, ожидания или получения значения из std::future. Эти методы могут заблокировать выполнение до получения результата асинхронной операции.
- Когда асинхронная операция готова к отправке результата её создателю, она может сделать это, изменив shared state, которое связано с std::future создателя.