Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовой проект

по курсу

«Операционные системы»

III Семестр

Студент: Шляхтуров А.В
Группа: М8О-201Б-22
Преподаватель: Миронов Е. С.
Вариант 39
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2023

Цель работы.

Создание планировщика DAG*'а «джобов» (jobs)**

Задание

По конфигурационному файлу в формате yaml, json или ini принимает спроектированный DAG джобов и проверяет на корректность: отсутствие циклов, наличие только одной компоненты связанности, наличие стартовых и завершающих джоб. Структура описания джоб и их связей произвольная. При завершении джобы с ошибкой, необходимо прервать выполнение всего DAG'а и всех запущенных джоб.

Решение

Для того, чтобы получать информацию от файла .ini будем использовать библиотеку boost, в которой есть инструменты для парсинга ini файлов. В ini файле будем хранить наш граф таким образом:

```
[jobs]

job1_dependencies =
job2_dependencies = job1
job3_dependencies = job2
job4_dependencies = job3
job5_dependencies = job4
job6_dependencies = job5
```

Переменные – это ноды нашего графа, а значения этих переменных являются спиками вершин, от которых зависят ноды.

Парсер, встроенный в библиотеку boost представляет ini файл в виде дерева.

Мы будем проверять этот направленный граф на связность, наличие циклов и начальных и конечных задач.

Код

```
#include <iostream>
#include <fstream>
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/ini_parser.hpp>
#include <set>
```

```cpp
#include <vector>
#include <string>
#include <queue>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>

// ОБРАБОТКА ЗАДАЧ И ЕЕ ПРЕДШЕСТВЕННИКОВ
//  A зависит от B, B зависит от C и D, D зависит от E
// {"A", {"B"}},
// {"B", {"C", "D"}},
// {"D", {"E"}} ->>> C, E, D, B, A
void execute_task(const std::string &task, const std::unordered_map<std::string,
std::vector<std::string>> &predecessors,
                  std::unordered_set<std::string> &processed, std::vec-
tor<std::string> &execution_order)
{
    // Заносим рассматриваемую ноду в список обработанных нод
    processed.insert(task);
    // Проверяем, есть ли цель task среди ключей
    if (predecessors.count(task) > 0)
    {
        // Извлекаем вектор узлов предшественников, от которых зависит наша цель
task
        const std::vector<std::string> &task_predecessors = predecessors.at(task);

        for (const auto &predecessor : task_predecessors)
        {
            if (processed.count(predecessor) == 0)
            {
                execute_task(predecessor, predecessors, processed, execution_or-
der);
                // Рекурсивно находим тех, у кого нет предшественников, и заносим в
конец списка order
            }
        }
    }
    execution_order.push_back(task);
}

// ПРОВЕРКА НА ЦИКЛИЧЕСКИЕ ЗАВИСИМОСТИ

bool hasCycleUtil(const std::unordered_map<std::string, std::vector<std::string>>
&graph, const std::string &node, std::unordered_set<std::string> &visited, std::un-
ordered_set<std::string> &recStack)
{
    // Помечаем текущую ноду как посещённую и добавляем в стек рекурсии
    visited.insert(node);
    recStack.insert(node);

    // Рекурсивно проверяем все зависимости текущей ноды
```

```
    for (const auto &neighbor : graph.at(node))
    {
        // Если сосед уже был посещён и находится в стеке рекурсии, то цикл
обнаружен
        if (recStack.find(neighbor) != recStack.end())
        {
            return true;
        }
        // Если сосед ещё не был посещён, то проверяем его
        else if (visited.find(neighbor) == visited.end())
        {
            if (hasCycleUtil(graph, neighbor, visited, recStack))
            {
                return true;
            }
        }
    }

    // Убираем текущую ноду из стека рекурсии
    recStack.erase(node);

    return false;
}

// ПРОВЕРКА СТРУКТУРЫ ГРАФА ЗАДАЧ

bool hasCycle(const boost::property_tree::ptree &config_tree)
{

    std::unordered_map<std::string, std::vector<std::string>> dependencies_map;

    for (const auto &job : config_tree.get_child("jobs"))
    {
        std::string jobName = job.first;
        if (jobName.find("_dependencies") != std::string::npos)
        {
            jobName = jobName.substr(0, jobName.find("_dependencies"));
        }
        std::string dependencies_str = job.second.data();
        std::vector<std::string> dependencies;
        std::istringstream iss(dependencies_str);
        std::string dependency;
        while (std::getline(iss, dependency, ','))
        {
            dependency.erase(std::remove_if(dependency.begin(), dependency.end(),
::isspace), dependency.end());
            if (!dependency.empty())
            {
                dependencies.push_back(dependency);
            }
        }
```

```
        dependencies_map[jobName] = dependencies;
    }

    std::unordered_set<std::string> visited;  // Множество посещённых нод
    std::unordered_set<std::string> recStack; // Стек рекурсии для обнаружения
циклов

    // Проходим по каждой ноде графа
    for (const auto &entry : dependencies_map)
    {
        const std::string &node = entry.first;
        // Если нода ещё не была посещена, проверяем наличие циклов, начиная с неё
        if (visited.find(node) == visited.end())
        {
            if (hasCycleUtil(dependencies_map, node, visited, recStack))
            {
                return true; // Если цикл обнаружен, возвращаем true
            }
        }
    }

    return false; // Если циклы не обнаружены, возвращаем false
}

bool dfs(const std::unordered_map<std::string, std::vector<std::string>> &graph,
std::unordered_set<std::string> &visited, const std::string &node)
{
    visited.insert(node);

    for (const auto &neighbor : graph.at(node))
    {
        if (visited.find(neighbor) == visited.end())
        {
            dfs(graph, visited, neighbor);
        }
    }

    return visited.size() == graph.size();
}

// Проверка на единственность компоненты связности обходом в глубину

bool is_single_component(const boost::property_tree::ptree &config_tree)
{
    std::unordered_map<std::string, std::vector<std::string>> graph;

    std::unordered_map<std::string, std::vector<std::string>> dependencies_map;

    for (const auto &job : config_tree.get_child("jobs"))
    {
        std::string jobName = job.first;
```

```
        if (jobName.find("_dependencies") != std::string::npos)
        {
            jobName = jobName.substr(0, jobName.find("_dependencies"));
        }
        std::string dependencies_str = job.second.data();
        std::vector<std::string> dependencies;
        std::istringstream iss(dependencies_str);
        std::string dependency;
        while (std::getline(iss, dependency, ','))
        {
            dependency.erase(std::remove_if(dependency.begin(), dependency.end(),
::isspace), dependency.end());
            if (!dependency.empty())
            {
                dependencies.push_back(dependency);
            }
        }
        dependencies_map[jobName] = dependencies;
    }
    std::unordered_set<std::string> visited;
    if (dependencies_map.empty())
        return false;

    const auto &start_node = dependencies_map.begin()->first;

    return dfs(dependencies_map, visited, start_node);
}

// Проверка наличия начальных и конечных задач
bool has_start_and_finish_tasks(const boost::property_tree::ptree &config_tree)
{
    std::set<std::string> tasks;
    std::set<std::string> tasks_with_deps;
    std::set<std::string> tasks_that_are_deps;

    for (const auto &entry : config_tree.get_child("jobs"))
    {
        const std::string &task = entry.first.substr(0, entry.first.find("_depend-
encies"));
        tasks.insert(task);

        std::string deps = entry.second.data();
        std::istringstream deps_stream(deps);
        std::string dep;
        while (std::getline(deps_stream, dep, ','))
        {
            if (!dep.empty())
            {
                dep.erase(std::remove_if(dep.begin(), dep.end(), ::isspace),
dep.end());
                tasks_with_deps.insert(dep);
```

```
                tasks_that_are_deps.insert(task);
            }
        }
    }

    std::set<std::string> initial_tasks;
    std::set_difference(tasks.begin(), tasks.end(),
                        tasks_with_deps.begin(), tasks_with_deps.end(),
                        std::inserter(initial_tasks, initial_tasks.end()));

    std::set<std::string> final_tasks;
    std::set_difference(tasks.begin(), tasks.end(),
                        tasks_that_are_deps.begin(), tasks_that_are_deps.end(),
                        std::inserter(final_tasks, final_tasks.end()));



    // Если есть ноды, которые ни от кого не зависят, и есть ноды, которые ни от
кого не зависят
    return !initial_tasks.empty() && !final_tasks.empty();
}

int main(int argc, char *argv[])
{
    // Если не передали имя файла ini
    if (argc < 2)
    {
        std::cerr << "Usage: " << argv[0] << " <workflow_config.ini>" << std::endl;
        return 1;
    }

    std::string config_filename = argv[1];
    boost::property_tree::ptree workflow_config;
    try
    {
        boost::property_tree::ini_parser::read_ini(config_filename, workflow_con-
fig);
    }
    catch (const boost::property_tree::ini_parser_error &e)
    {
        std::cerr << "Failed to parse config: " << e.what() << std::endl;
        return 1;
    }

    if (hasCycle(workflow_config))
    {
        std::cerr << "Error: Workflow contains cycles." << std::endl;
        return 1;
    }

    if (!has_start_and_finish_tasks(workflow_config))
```

```
    {
        std::cerr << "Error: Workflow lacks initial or final tasks." << std::endl;
        return 1;
    }

    if (!is_single_component(workflow_config))
    {
        std::cerr << "Error: Workflow is not fully connected." << std::endl;
        return 1;
    }



    std::cout << "Workflow is structured correctly." << std::endl;

    std::unordered_map<std::string, std::vector<std::string>> dependencies_map;
    for (const auto &job : workflow_config.get_child("jobs"))
    {
        std::string jobName = job.first;
        if (jobName.find("_dependencies") != std::string::npos)
        {
            jobName = jobName.substr(0, jobName.find("_dependencies"));
        }
        std::string dependencies_str = job.second.data();
        std::vector<std::string> dependencies;
        std::istringstream iss(dependencies_str);
        std::string dependency;
        while (std::getline(iss, dependency, ','))
        {
            dependency.erase(std::remove_if(dependency.begin(), dependency.end(),
::isspace), dependency.end());
            if (!dependency.empty())
            {
                dependencies.push_back(dependency);
            }
        }
        dependencies_map[jobName] = dependencies;
    }

    std::unordered_set<std::string> visitedJobs;
    std::vector<std::string> result;
    for (const auto &job : dependencies_map)
    {
        if (visitedJobs.count(job.first) == 0)
        {
            execute_task(job.first, dependencies_map, visitedJobs, result);
        }
    }

    // std::reverse(result.begin(), result.end());
    for (const auto &job : result)
```

```
    {
        if (job.find("_dependencies") == std::string::npos)
        {
            std::cout << "Job done: " << job << std::endl;
        }
    }

    std::cout << "All jobs done" << std::endl;

    return 0;
}
```

Выводы

Работа над курсовым проектом была для меня очень полезна. Я реализовал планировщик задач, принимающий направленный граф джобов и проверяющий их на корректность. Для того, чтобы проверять граф на ацикличность и компоненты связности пришлось вспомнить алгоритмы обхода графа в глубину и в ширину, что, несомненно, было большим плюсом. Благодаря работе над курсовым проектом я также получил опыт работы с библиотекой boost, а именно с такими ее инструментами, как property_tree и ini_parser. Я получил новые навыки и освежил некоторые старые знания в ходе работы над курсовым проектом по предмету «Операционные системы».