Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу

«Операционные системы»

Студент: Дудовцев Андрей Андреевич
Группа: М8О-208Б-22
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2024

Содержание

1. Репозиторий
2. Цель работы
3. Задание
4. Описание работы программы
5. Исходный код
6. Консоль
7. Примеры конфигурационного файла
8. Выводы

Репозиторий

AndreyDdvts/OS_LABS (github.com)

Цель работы

Приобретение практических навыков в:

- Использовании знаний, полученных в течении курса
- Проведение исследования в выбранной предметной области

Задание

На языке C++ написать программу, которая:

1. 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;
2. При завершении джобы с ошибкой, необходимо прервать выполнение всего DAG'а и всех запущенных джоб;
3. Джобы должны запускаться максимально параллельно. Должны быть ограниченны параметром – максимальным числом одновременно выполняемых джоб

Описание работы программы

Мой вариант задания заключается в создании планировщика «процессовзадач» по конфигурационному файлу в формате ini. Для обработки данного формата я использовал парсер inipp.h.

Моя курсовая работа состоит из следующих файлов:

- dag.hpp/dag.cpp - Класс DAG содержит вектор всех джобов и граф с их зависимостями;
- executor.hpp/executor.cpp - Исполнитель DAG. Для каждого исполняемого процесса создается отдельный поток, который ждет, когда дочерний процесс выполнится. Далее передается сообщение о

выполнении в класс Pipe, родительский процесс читает это сообщение и продолжает выполнять следующие задачи. Использование примитива синхронизации, очереди процессов, а также списка зависимостей позволяет отслеживать, какие процессы могут выполняться параллельно;

- graph.hpp/graph.cpp - Здесь реализован сам граф, в котором содержатся джобы в нужном порядке, а также его проверка на наличие циклов, используя алгоритм поиска в глубину (Dfs);
- parser.hpp/parser.cpp - Парсер DAG'а из ini файла.

Исходный код

```
========================= dag.hpp =========================

#pragma once

#include <iostream>
#include <vector>

#include "graph.hpp"

struct Job {
    std::string name, path;
};

class DAG {
private:
    std::vector<Job> jobs;
    Graph graph;

public:
    DAG() = delete;
    DAG(const std::vector<Job> &_jobs, const Graph &_graph);
```

```
    const std::vector<Job> &GetJobs() const;
    const Graph &GetGraph() const;
};

========================== executor.hpp ==========================

#pragma once

#include <thread>
#include <queue>
#include <set>
#include <mutex>
#include <condition_variable>
#include <atomic>
#include <unistd.h>
#include <wait.h>

#include "dag.hpp"

int StartProcess(const std::string &path);

class Pipe {
private:
    std::queue<size_t> q;
    std::mutex mtx;
    std::condition_variable cv;
public:
    void Push(size_t);
    std::vector<size_t> Pop();
};

class Executor {
private:
    DAG &dag;
    size_t freeThreads;
```

```
    void ExecuteJob(size_t id, Job job, Pipe *pipe);
public:
    Executor(DAG &_dag);
    void Execute(size_t threadCount);

};

========================== graph.hpp ==========================

#pragma once

#include <iostream>
#include <vector>
#include <map>

class Graph {
public:
    using Node = size_t;

    Graph(size_t N) : edges(N) { }

    size_t NodeCount() const;
    void AddEdge(Node from, Node to);
    bool CheckCycles() const;

    const std::vector<std::vector<Node> > &GetEdges() const;
private:
    std::vector<std::vector<Node> > edges;

    bool Dfs(Node current, std::vector<int> &visited) const;
};

========================== parser.hpp ==========================
```

```
#pragma once

#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>

#include "dag.hpp"

#include "inipp.h"

DAG Parse(const std::string &path);

========================== dag.cpp ==========================

#include "dag.hpp"

DAG::DAG(const std::vector<Job> &_jobs, const Graph &_graph) :
jobs(_jobs), graph(_graph) {
    if (graph.NodeCount() != jobs.size()) {
        throw std::logic_error("Nodes count != jobs count");
    }
    if (!graph.CheckCycles()) {
        throw std::logic_error("Graph has cycle");
    }
}

const std::vector<Job> &DAG::GetJobs() const {
    return jobs;
}

const Graph &DAG::GetGraph() const {
    return graph;
}
```

```
========================== executor.cpp ==========================

#include "executor.hpp"

int StartProcess(const std::string &path) {
    int pid = fork();
    if (pid == -1) {
        throw std::logic_error("Can't fork");
    }
    else if (pid == 0) {
        if (execl(path.c_str(), path.c_str(), nullptr) == -1) {
            throw std::logic_error("Can't exec");
        }
    } else {
        int status;
        waitpid(pid, &status, 0);
        return status;
    }
    return 0;
}

void Pipe::Push(size_t id) {
    {
        std::lock_guard<std::mutex> lk(mtx);
        q.push(id);
    }
    cv.notify_one();
}

std::vector<size_t> Pipe::Pop() {
    std::vector<size_t> result;
    {
        std::unique_lock<std::mutex> lk(mtx);
        if (q.empty()) {
            cv.wait(lk);
```

```
        }
        while (!q.empty()) {
            result.push_back(q.front());
            q.pop();
        }
    }
    return result;
}

Executor::Executor(DAG &_dag) : dag(_dag) { }

void Executor::ExecuteJob(size_t id, Job job, Pipe *pipe) {
    int result = StartProcess(job.path);
    if (result != 0) {
        exit(EXIT_FAILURE);
    } else {
        pipe->Push(id);
    }
}

void Executor::Execute(size_t threadCount) {
    freeThreads = threadCount;
    size_t count = dag.GetJobs().size();
    size_t iter = count;
    std::vector<size_t> toExecute;
    Pipe pipe;
    std::vector<int> numOfDeps(count, 0);

    for (size_t from = 0; from < count; ++from) {
        for (const auto &to : dag.GetGraph().GetEdges()[from]) {
            numOfDeps[to]++;
        }
    }

    for (size_t id = 0; id < count; ++id) {
```

```
        if (numOfDeps[id] == 0) {
            toExecute.push_back(id);
            numOfDeps[id] = -1;
        }
    }

    while (iter != 0) {
        while (!toExecute.empty() && freeThreads != 0) {
            size_t id = toExecute[toExecute.size() - 1];
            std::thread t(&Executor::ExecuteJob, this, id,
dag.GetJobs()[id], &pipe);
            t.detach();
            freeThreads--;
            toExecute.pop_back();
        }

        std::vector<size_t> result = pipe.Pop();
        for (const auto &id : result) {
            freeThreads++;
            iter--;
            for (const auto &to : dag.GetGraph().GetEdges()[id]) {
                numOfDeps[to]--;
            }
        }

        for (size_t id = 0; id < count; ++id) {
            if (numOfDeps[id] == 0) {
                toExecute.push_back(id);
                numOfDeps[id] = -1;
            }
        }
    }
}
```

========== graph.cpp ==========

```
#include "graph.hpp"

bool Graph::Dfs(Node current, std::vector<int> &visited) const {
    visited[current] = 1;
    for (const auto& to : edges[current]) {
        if (visited[to] == 1) {
            return true;
        } else if (visited[to] == 0) {
            bool result = Dfs(to, visited);
            if (result) {
                return true;
            }
        }
    }
    visited[current] = 2;
    return false;
}

size_t Graph::NodeCount() const {
    return edges.size();
}

void Graph::AddEdge(Node from, Node to) {
    edges[from].push_back(to);
}

bool Graph::CheckCycles() const {
    std::vector<int> visited(NodeCount(), 0);
    for (Node node = 0; node < NodeCount(); ++node) {
        if (visited[node] == 0) {
            if (Dfs(node, visited)) {
                return false;
            }
        }
```

```
    }
    return true;
}

const std::vector<std::vector<Graph::Node> > &Graph::GetEdges() const
{
    return edges;
}

========================== parser.cpp ==========================

#include "parser.hpp"
#include <iostream>

DAG Parse(const std::string &path) {
    inipp::Ini<char> ini;
    std::ifstream is(path);
    ini.parse(is);

    std::string pathToBin, rawJobs, rawDependencies, rawCount;
    size_t count;

    inipp::get_value(ini.sections["general"], "bin_path", pathToBin);
    inipp::get_value(ini.sections["jobs"], "count", rawCount);
    inipp::get_value(ini.sections["jobs"], "jobs", rawJobs);
    inipp::get_value(ini.sections["dependencies"], "dependencies",
rawDependencies);

    count = std::stoi(rawCount);
    std::vector<Job> jobs;
    Graph graph(count);
    std::map<std::string, size_t> jobsToId;

    std::stringstream ss(rawJobs);
    std::string current;
```

```
    while (getline(ss, current, ',')) {
        std::string name(current.begin() + 1, current.end());
        getline(ss, current, ',');
        std::string path(current.begin(), current.end() - 1);
        path = pathToBin + "/" + path;
        jobs.push_back({name, path});
        jobsToId[name] = jobs.size() - 1;
    }

    ss = std::stringstream(rawDependencies);
    while (getline(ss, current, ',')) {
        std::string req(current.begin() + 1, current.end());
        getline(ss, current, ',');
        std::string target(current.begin(), current.end() - 1);
        graph.AddEdge(jobsToId[req], jobsToId[target]);
    }

    return DAG(jobs, graph);
}

========================== main.cpp ==========================

#include <iostream>
#include <fstream>
#include <chrono>

#include "inipp.h"

#include "parser.hpp"
#include "executor.hpp"

// bash: export
PATH_TO_CONFIG="/home/qwz/OS_LABS/coursework/data/ex1/config.ini"

int main(int argc, char ** argv) {
```

```
    size_t threadCount = 4;
    if (argc > 1) {
        threadCount = std::atoi(argv[1]);
    }

    DAG dag = Parse(std::string(getenv("PATH_TO_CONFIG")));
    Executor exec(dag);

     auto begin = std::chrono::high_resolution_clock::now();
    exec.Execute(threadCount);
     auto end = std::chrono::high_resolution_clock::now();
     int time =
std::chrono::duration_cast<std::chrono::milliseconds>(end -
begin).count();
     std::cout << "Time for " << threadCount << " threads: " << time
<< std::endl;
    return 0;
}
```

Консоль

```
job2 started ====
job2 finished ===
job1 started ====
job1 finished ===
job3 started ====
job3 finished ===
Time for 1 threads: 6826
qwz@qwz-VirtualBox:~/OS_LABS/build/coursework$ ./cw_main 2
job2 started ====
job1 started ====
job2 finished ===
job1 finished ===
job3 started ====
```

```
job3 finished ===
Time for 2 threads: 2007
qwz@qwz-VirtualBox:~/OS_LABS/build/coursework$./cw_main 4
job2 started ====
job1 started ====
job2 finished ===
job1 finished ===
job3 started ====
job3 finished ===
Time for 4 threads: 2005
```

Примеры конфигурационного файла

В конфигурационном файле есть три секции:

1. general - Переменная bin_path, содержащая путь к джобам;
2. jobs - Переменная количества джобов и сами джобы в виде пар (name,path);
3. dependencies - Переменная зависимостей в виде пар (required,target).

Пример 1.

```
[general]

bin_path=/home/qwz/OS_LABS/coursework/data/ex1/bin

[jobs]

count=3
jobs=(job1,job1),(job2,job2),(job3,job3)



[dependencies]

dependencies=(job1,job3),(job2,job3)
```

Пример 2.

```
[general]
```

```
bin_path=/home/qwz/OS_LABS/coursework/data/ex2/bin

[jobs]

count=8
jobs=(job1,job1),(job2,job2),(job3,job3),(job4,job4),(job5,job5),(job6,job6),
(job7,job7),(job8,job8)

[dependencies]

dependencies=(job1,job6),(job2,job6),(job3,job6),(job4,job7),(job5,job7),(job
6,job8),(job7,job8)
```

Выводы

В результате выполнения данной курсовой работы была реализована программа на C++, которая по конфигурационному файлу в формате ini принимает DAG джобов и планирует их выполнение, учитывая заданные зависимости. Она успешно справляется с поставленными задачами: проверяет конфигурационный файл на наличие ошибок, таких как циклы, наличие одной компоненты связанности и наличие стартовой/завершающих джобов. Также было уделено внимание максимальной параллельности выполнения джобов. Реализован механизм оптимизации, позволяющий эффективно распределять задачи для лучшего использования ресурсов и сокращения времени выполнения, что видно при запуске программы. В итоге я приобрел практические навыки в использовании знаний, полученных в течении курса, и проведении исследования в выбранной предметной области, которые обязательно пригодятся в будущем.