Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Курсовой проект по курсу**

**«Операционные системы»**

Студент: Ибрагимов Далгат Магомедаливеич

Группа: М8О-208Б-22

Вариант: 31

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2023

**Содержание**

1. Репозиторий

2. Постановка задачи

3. Общие сведения о программе

4. Общий метод и алгоритм решения

5. Исходный код

6. Демонстрация работы программы

7. Выводы

**Репозиторий**

<https://github.com/L0ckR/OS_LABS>

**Постановка задачи**

На языке C\C++ написать программу, которая по конфигурационному файлу в формате json принимает спроектированный DAG джобов и проверяет на корректность: отсутствие циклов, наличие только одной компоненты связанности, наличие стартовых и завершающих джоб.

**Общие сведения о программе**

Программа написана на C++ на операционную систему Linux. Для парсинга YAML файлов была использована библиотека jbeder/yaml-cpp (<https://github.com/jbeder/yaml-cpp>).

**Общий метод и алгоритм решения**

Программа парсит конфигурационный файл и на его основе строит DAG джобов, проверяет его корректность, и, если все нормально, начинает его выполнение. В каждый момент времени мы храним таблицу, в которой для каждой джобы указано, от скольки еще не выполненных джоб она зависит, после каждой выполненной джобы мы обновляем таблицу, если мы видим что у какой то джобы счетчик равен нулю, она перемещается в очередь на выполнение. Для выполнения джобы создается отдельный поток, который создает дочерний процесс и ждет его завершения.

**Исходный код**

**========================== job\_dag.hpp ==========================**

**#pragma once**

**#include <iostream>**

**#include <vector>**

**#include <map>**

**#include <set>**

**#include <fstream>**

**#include "yaml-cpp/yaml.h"**

**namespace cp {**

**class TDagJobExecutor;**

**struct TJob {**

**std::string name, path;**

**};**

**class TJobDag {**

**private:**

**using TMapStringToStrings = std::map<std::string, std::vector<std::string>>;**

**// Common rule :**

**// if the pair represents dependency then first element is what REQUIRED and the second one is TARGET**

**// Map of jobs**

**std::map<std::string, TJob> jobs;**

**// Map of dependencies. dep["name"] gives us vector of jobs REQUIRED to do job "name"**

**TMapStringToStrings dep;**

**// Inversed dep. dep^-1**

**TMapStringToStrings rdep;**

**std::vector<std::string> mutexes;**

**// Map of mutexes, each job own : mutOwn[j] is vector of mutexes, which j own**

**TMapStringToStrings mutOwn;**

**// 0 - not-visited, 1 - in current route, 2 - visited and not in current route**

**// returns true if have loops, false - otherwise**

**static bool Dfs(const std::string &v,**

**std::map<std::string, int>& visited,**

**TMapStringToStrings& dep);**

**static bool CheckCorrectness(TJobDag &dag);**

**static TMapStringToStrings Inverse(TMapStringToStrings &map);**

**public:**

**friend class TDagJobExecutor;**

**TJobDag() = default;**

**TJobDag(const std::vector<TJob>& jobs,**

**const std::vector<std::pair<std::string, std::string>>& deps,**

**const std::vector<std::string>& \_mutexes = {},**

**const TMapStringToStrings& \_mutOwn = {});**

**};**

**class YAMLParser {**

**public:**

**static TJobDag Parse(const std::string &pathToFile);**

**};**

**} // namespace cp**

**========================== job\_exec.hpp ==========================**

**#pragma once**

**#include <iostream>**

**#include <thread>**

**#include <mutex>**

**#include <condition\_variable>**

**#include <optional>**

**#include <unistd.h>**

**#include <sys/wait.h>**

**#include "job\_dag.hpp"**

**namespace cp {**

**class TSystem {**

**public:**

**static int Exec(const std::string& path);**

**};**

**struct LogStack {**

**// Stack of completed jobs**

**std::vector<std::string> completed;**

**size\_t wasRead = 0;**

**std::mutex mut;**

**std::condition\_variable cv;**

**void Push(const std::string &str);**

**};**

**class TBasicExecutor {**

**private:**

**const constexpr static size\_t STANDART\_MAX\_PROC\_COUNT = 4;**

**size\_t maxProcCount;**

**LogStack \* log;**

**size\_t procCount = 0;**

**std::mutex mpcountMutex;**

**std::condition\_variable mpcountCondVar;**

**void RawExecute(const std::string &name, const std::string &path, LogStack \*log);**

**public:**

**TBasicExecutor(LogStack \*\_log) : maxProcCount(STANDART\_MAX\_PROC\_COUNT), log(\_log) { }**

**TBasicExecutor(size\_t mpCount, LogStack \*\_log) : maxProcCount(mpCount), log(\_log) { }**

**void Execute(const std::string &path, const std::string &name);**

**};**

**class TDagJobExecutor {**

**private:**

**const static size\_t STANDART\_MAX\_PROC\_COUNT = 4;**

**size\_t maxProcCount;**

**size\_t target, current;**

**size\_t procCount;**

**std::set<std::string> actuallyReadyToBeExecuted;**

**std::mutex mut; // procCount and sets**

**std::condition\_variable cv;**

**LogStack log;**

**TBasicExecutor ex;**

**public:**

**TDagJobExecutor() : maxProcCount(STANDART\_MAX\_PROC\_COUNT), ex(maxProcCount, &log) { }**

**TDagJobExecutor(size\_t mpCount) : maxProcCount(mpCount), ex(maxProcCount, &log) { }**

**bool Execute(TJobDag &dag);**

**};**

**}**

**========================== job\_dag.cpp ==========================**

**#include "job\_dag.hpp"**

**namespace cp {**

**bool TJobDag::Dfs(const std::string &v,**

**std::map<std::string, int>& visited,**

**TMapStringToStrings& dep) {**

**visited[v] = 1;**

**for (const auto& to : dep[v]) {**

**if (visited[to] == 1) {**

**return true;**

**} else if (visited[to] == 0) {**

**bool result = Dfs(to, visited, dep);**

**if (result) {**

**return result;**

**}**

**}**

**}**

**visited[v] = 2;**

**return false;**

**}**

**bool TJobDag::CheckCorrectness(TJobDag &dag) {**

**// First step (map keys is job.name)**

**for (const auto& p : dag.jobs) {**

**const auto& key = p.first;**

**const auto& job = p.second;**

**if (key != job.name) {**

**return false;**

**}**

**}**

**// Second step**

**for (const auto& p : dag.dep) {**

**if (dag.jobs.find(p.first) == dag.jobs.end()) {**

**return false;**

**}**

**for (const auto& i : p.second) {**

**if (dag.jobs.find(i) == dag.jobs.end()) {**

**return false;**

**}**

**}**

**}**

**// Third step (Absence of loops)**

**std::map<std::string, std::vector<std::string>> dep = dag.dep;**

**std::map<std::string, int> visited;**

**for (const auto& p : dep) {**

**visited[p.first] = 0;**

**}**

**for (const auto& p : dep) {**

**if (visited[p.first] == 0) {**

**if (Dfs(p.first, visited, dep)) {**

**return false;**

**}**

**}**

**}**

**return true;**

**}**

**TJobDag::TMapStringToStrings TJobDag::Inverse(TMapStringToStrings &map) {**

**TMapStringToStrings result;**

**for (const auto& p : map) {**

**for (const auto& target : p.second) {**

**result[target].push\_back(p.first);**

**}**

**}**

**return result;**

**}**

**TJobDag::TJobDag(const std::vector<TJob>& jobs,**

**const std::vector<std::pair<std::string, std::string>>& deps,**

**const std::vector<std::string>& \_mutexes,**

**const TMapStringToStrings& \_mutOwn) {**

**for (const auto& i : jobs) {**

**this->jobs[i.name] = i;**

**}**

**for (const auto& p : deps) {**

**dep[p.second].push\_back(p.first);**

**}**

**if (!CheckCorrectness(\*this)) {**

**throw std::logic\_error("Bad DAG");**

**}**

**mutexes = \_mutexes;**

**mutOwn = \_mutOwn;**

**rdep = Inverse(dep);**

**}**

**TJobDag YAMLParser::Parse(const std::string &pathToFile) {**

**YAML::Node yml = YAML::LoadFile(pathToFile);**

**std::string path\_to\_bins = yml["path\_to\_bins"].as<std::string>();**

**std::vector<TJob> jobs;**

**for (const auto& job : yml["jobs"]) {**

**std::string path = path\_to\_bins + job["path"].as<std::string>();**

**jobs.push\_back({job["name"].as<std::string>(), path});**

**}**

**std::vector<std::pair<std::string, std::string>> deps;**

**for (const auto& dep : yml["dependencies"]) {**

**deps.push\_back({dep["required"].as<std::string>(), dep["target"].as<std::string>()});**

**}**

**std::vector<std::string> muts;**

**std::map<std::string, std::vector<std::string>> mutOwn;**

**for (const auto& mut : yml["mutexes"]) {**

**muts.push\_back(mut["name"].as<std::string>());**

**for (const auto& owner : mut["who\_owns"]) {**

**mutOwn[owner.as<std::string>()].push\_back(mut["name"].as<std::string>());**

**}**

**}**

**return TJobDag(jobs, deps, muts, mutOwn);**

**}**

**}**

**========================== job\_exec.cpp ==========================**

**#include "job\_exec.hpp"**

**#include <atomic>**

**namespace cp {**

**int TSystem::Exec(const std::string& path) {**

**int pid = fork();**

**if (pid == 0) {**

**if (execl(path.c\_str(), path.c\_str(), nullptr) == -1) {**

**std::cout << "Can't exec\n";**

**}**

**} else if (pid == -1) {**

**throw std::logic\_error("Can't fork");**

**} else {**

**int status;**

**waitpid(pid, &status, 0);**

**return status;**

**}**

**return 0;**

**}**

**void LogStack::Push(const std::string &str) {**

**{**

**std::unique\_lock l(mut);**

**completed.push\_back(str);**

**}**

**cv.notify\_one();**

**}**

**void TBasicExecutor::Execute(const std::string &path, const std::string &name) {**

**std::thread t(&TBasicExecutor::RawExecute, this, name, path, log);**

**t.detach();**

**}**

**void TBasicExecutor::RawExecute(const std::string &name, const std::string &path, LogStack \*log) {**

**{**

**std::unique\_lock<std::mutex> lk(mpcountMutex);**

**if (maxProcCount == procCount) {**

**mpcountCondVar.wait(lk);**

**}**

**procCount++;**

**}**

**try {**

**int status = TSystem::Exec(path);**

**if (status != 0) {**

**exit(EXIT\_FAILURE);**

**}**

**} catch (...) {**

**exit(EXIT\_FAILURE);**

**}**

**{**

**std::unique\_lock<std::mutex> lk(mpcountMutex);**

**procCount--;**

**}**

**mpcountCondVar.notify\_one();**

**log->Push(name); // Warn about task completed**

**}**

**bool TDagJobExecutor::Execute(TJobDag &dag) {**

**target = dag.jobs.size();**

**current = 0;**

**if (target == 0) {**

**return true;**

**}**

**std::set<std::string\_view> executionQueue;**

**for (const auto& p : dag.jobs) {**

**executionQueue.insert(p.first);**

**}**

**std::map<std::string\_view, int> countOfDeps;**

**for (const auto& p : dag.dep) {**

**countOfDeps[p.first] = p.second.size();**

**}**

**// First layer**

**for (const auto& p : dag.jobs) {**

**if (countOfDeps[p.first] == 0) {**

**actuallyReadyToBeExecuted.insert(p.first);**

**executionQueue.erase(p.first);**

**}**

**}**

**// Mutexes handling**

**std::map<std::string, std::atomic<bool> > mutexState; // Owned - true, not owned - false**

**for (const auto& i : dag.mutexes) {**

**mutexState[i] = false;**

**}**

**// Execution**

**procCount = 0;**

**while (true) {**

**if (current == target) {**

**return true;**

**} else {**

**{**

**std::unique\_lock l2(mut);**

**std::vector<std::string> toErase;**

**for (const auto& job : actuallyReadyToBeExecuted) {**

**bool flag = true;**

**for (const auto& mutexOwn : dag.mutOwn[job]) {**

**if (mutexState[mutexOwn]) {**

**flag = false;**

**break;**

**}**

**}**

**if (!flag) {**

**continue;**

**}**

**for (const auto& mutexOwn : dag.mutOwn[job]) {**

**mutexState[mutexOwn] = true;**

**}**

**toErase.push\_back(job);**

**ex.Execute(dag.jobs[job].path, job);**

**}**

**for (const auto& job : toErase) {**

**actuallyReadyToBeExecuted.erase(job);**

**}**

**}**

**std::vector<std::string> completed;**

**{**

**std::unique\_lock l(log.mut);**

**if (log.wasRead == log.completed.size()) {**

**log.cv.wait(l);**

**}**

**for (size\_t i = log.wasRead; i < log.completed.size(); i++) {**

**completed.push\_back(log.completed[i]);**

**current++;**

**}**

**log.wasRead = log.completed.size();**

**}**

**for (const auto& job : completed) {**

**for (const auto& depend : dag.rdep[job]) {**

**countOfDeps[depend]--;**

**if (countOfDeps[depend] == 0) {**

**actuallyReadyToBeExecuted.insert(depend);**

**executionQueue.erase(depend);**

**}**

**}**

**for (const auto& mtx : dag.mutOwn[job]) {**

**mutexState[mtx] = false;**

**}**

**}**

**}**

**}**

**}**

**}**

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

**#include <iostream>**

**#include <fstream>**

**#include "job\_dag.hpp"**

**#include "job\_exec.hpp"**

**using namespace cp;**

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

**if (argc < 2) {**

**std::cerr << "Missing arguments : path to config file\n";**

**exit(EXIT\_FAILURE);**

**}**

**int maxJobs = 4;**

**if (argc > 2) {**

**maxJobs = std::atoi(argv[2]);**

**}**

**std::string pathToConfig(argv[1]);**

**TJobDag dag = YAMLParser::Parse(pathToConfig);**

**TDagJobExecutor executor(maxJobs);**

**executor.Execute(dag);**

**std::cout << "Execution finished!\n";**

**}**

**Демонстрация работы программы**

****

**Выводы**

В ходе выполнения курсового проекта я получил базовое представление о работе систем сборки типа make. Я закрепил навыки работы с многопоточными программами.