Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Операционные системы»

Создание планировщика DAG’а «джобов» (jobs)

Студент: Кунавин К. В.
Преподаватель: Миронов Е. С.
Группа: М8О-203Б-23
Дата:
Оценка:
Подпись:

Москва, 2024

Условие

DAG - Directed acyclic graph. Направленный ациклический граф.

Джоб (Job) – процесс, который зависит от результата выполнения других процессов (если он не стартовый), которые исполняются до него в DAG, и который порождает данные от которых может быть зависят другие процессы, которые исполняются после него в DAG (если он не завершающий).

На языке C\C++ написать программу, которая:

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.

(на оценку 4) Джобы должны запускаться максимально параллельно. Должны быть ограниченны параметром – максимальным числом одновременно выполняемых джоб.

(на оценку 5) Реализовать для джобов один из примитивов синхронизации мьютекс\семафор\барьер. То есть в конфиге дать возможность определять имена семафоров (с их степенями)\мьютексов\барьеров и указывать их в определение джобов в конфиге. Джобы указанные с одним мьютексом могут выполняться только последовательно (в любом порядке допустимом в DAG). Джобы указанные с одним семафором могут выполняться параллельно с максимальным числом параллельно выполняемых джоб равным степени семафору. Джобы указанные с одним барьером имеют следующие свойство – зависимые от них джобы начнут выполняться не раньше того момента времени, когда выполнятся все джобы с указанным барьером.

Задание

Ini\Barrier

Метод решения

Весь основной функционал по выполнению DAG инкапсулирован в классе TDagExecutor. Он хранит:

- **std::map<int, TJob> Jobs_**: все «джобы», где TJob содержит:
  - JobId,
  - BarrierName,
  - списки Dependencies и Children,
  - атомик RemainDeps (сколько зависимостей осталось невыполненными),
  - атомики IsFinished, IsSuccess.
- **std::map<std::string, TBarrierGroup> BarrierGroups_**: каждый барьер (по имени) содержит:

- Remaining (сколько джоб в этом барьере ещё не завершилось),
- pthread_mutex_t и pthread_cond_t.

- **std::queue<int> ReadyQueue_**: очередь «готовых» к запуску джоб.
- Параметр **MaxParallel_**: максимальное число параллельных джоб.
- **StopExecution_** (atomic bool): если хотя бы одна джоба упала — ставим true.
- Механизмы синхронизации:
  - pthread_mutex_t QueueMutex_ и pthread_cond_t QueueCond_ для управления ReadyQueue_.
  - глобальный мьютекс g_PrintMutex для потокобезопасного вывода.

2.2. Процесс выполнения

1. **Чтение INI**: метод ReadIni() парсит [Jobs], [Edges], [Parallel].
   - Для [Jobs]: создаём TJob, записываем в Jobs_.
   - Для [Edges]: связываем Children и Dependencies.
   - [Parallel] задаёт MaxParallel_.
   - Вычисляем RemainDeps и создаём «барьеры» — BarrierGroups_, если есть BarrierName.
2. **Проверка DAG**:
   - CheckCycle() — DFS и «рекурсивный стек» для обнаружения цикла.
   - CheckStartFinish() — проверяет, есть ли хотя бы одна джоба без зависимостей (start) и хотя бы одна джоба без детей (finish).
3. **Запуск**: метод RunDAG():
   - Добавляем в ReadyQueue_ все джобы, у которых RemainDeps == 0 (стартовые), **без** проверки барьера (иначе они никогда не запустятся).
   - В цикле диспетчер:
     - Ждёт, пока появятся «готовые» джобы **или** освободятся потоки **или** сработает StopExecution_.
     - Запускает новые потоки, пока не достигнем MaxParallel_.
4. **Выполнение джобы**: метод JobRunner() (статическая функция, запускаемая в pthread_create()):
   - Имитирует работу (sleep 1..3 сек).
   - Случайным образом «падает» для теста (нечётные jobId в 1/5 случаев).
   - Если упала — StopExecution_ = true.
   - По окончании вызывает BarrierArrive(), уменьшая Remaining в группе, и если дошли до 0 — pthread_cond_broadcast().
   - Если всё успешно, «разблокируем» дочерние джобы (их RemainDeps уменьшаем) и, если барьер у потомка разблокирован, добавляем их в ReadyQueue_.
   - Уменьшаем счётчик текущих потоков CurrentActiveThreads_.
   - Сигналим диспетчеру pthread_cond_broadcast(&QueueCond_).

3. Используемые библиотеки и инструменты

1. **POSIX Threads (pthreads)**:
    - pthread_create(), pthread_detach(),
    - мьютексы (pthread_mutex_t),
    - условные переменные (pthread_cond_t).
2. **C++ STL**:
    - std::map, std::vector, std::queue, std::atomic, std::set.
    - std::string, std::ostringstream, std::ifstream для парсинга INI.
3. **Google Test (GTest)**: для модульного тестирования.
4. **CMake**: для сборки проекта.

4. Реализация барьера (Barrier)

В классическом POSIX pthread_barrier_t нужно заранее знать точное число потоков, которые будут «дожидаться» барьера, и все они «встречаются» в одном месте.
В данной задаче другая семантика:

- «Барьер» по имени **B** объединяет **несколько джоб**.
- Зависимая джоба может стартовать только тогда, когда **все** джобы барьера **B** завершились.

Поэтому мы реализуем **«countdown barrier»**: для каждой группы (barrier name) хранится:

- **Remaining**: число джоб из этого барьера, которые ещё не завершились.
- Когда джоба заканчивается, вызывается BarrierArrive(B), внутри которого Remaining уменьшается. Если достигает 0, делаем pthread_cond_broadcast().
- Любая зависимая джоба, если у неё барьер **B**, стартует только если Remaining == 0. Иначе она «ждёт» (фактически, не попадает в ReadyQueue_ или ждёт, пока диспетчер не увидит, что барьер разблокирован).

Таким образом, «countdown barrier» реализован через pthread_mutex_t + pthread_cond_t.

5. Тестирование

1. **ParseWithoutCycle**
    - Создаёт INI-файл без цикла (пример из условия: job=1,2,3 и т.д.)
    - Проверяет, что парсинг прошёл (ReadIni() вернул true).
    - Проверяет, что в Jobs_ ожидаемое количество джоб (6).
    - Проверяет, что CheckCycle() даёт false, CheckStartFinish() даёт true.
    - Проверяет, что MaxParallel_ == 4 (как в [Parallel]).
2. **ParseWithCycle**
    - Создаёт INI-файл, где есть цикл (например, 2->3 и 3->2).
    - Убеждаемся, что CheckCycle() выдаёт true.

3. **ParseEmptyConfig**
    - Создаёт пустой файл (нет [Jobs], [Edges]).
    - ReadIni() откроет файл, но джоб не будет. CheckStartFinish() => false.
4. **ParseMultipleComponents**
    - Создаём INI-файл, где есть несколько вершин, но **часть** из них не связана с остальными (например, джобы 1->2, а джоба 3 сама по себе).
    - CheckCycle() будет false (ведь 1->2 не образует цикл, а 3 вообще отдельно).
    - CheckStartFinish() может дать true, если у всех вершин нет конфликтов (1 — start, 2 — finish, 3 — тоже start/finish «сам себе»).
    - **CheckSingleComponent()** (новая функция) вернёт false, т.к. мы обнаружим, что есть как минимум две компоненты.
5. **BarrierCheck** (пример расширенного теста на барьеры)
    - Создаём небольшой INI-файл, где job=1 и job=2 принадлежат барьеру B1, а job=3 зависит от обоих (1->3, 2->3).
    - Проверяем, что после парсинга CheckCycle() даёт false, CheckStartFinish() даёт true (start=1,2; finish=3), и в Jobs_ действительно отразилась зависимость 3 от 1 и 2.

Код программы

dag.h

```
#ifndef DAG_H
#define DAG_H

#include <pthread.h>
#include <atomic>
#include <map>
#include <queue>
#include <string>
#include <vector>
#include <set>
#include <iostream>
#include <fstream>

// ----------------------------------------------------------------------------
// Глобальный мьютекс для потокобезопасного вывода
// ----------------------------------------------------------------------------
extern pthread_mutex_t g_PrintMutex;

// ----------------------------------------------------------------------------
// Вспомогательная функция для «безопасного» вывода
// ----------------------------------------------------------------------------
void SafePrint(const std::string &message);

// ----------------------------------------------------------------------------
```

```
// Класс «джобы» (TJob) с пользовательским копированием,
// чтобы исправить ошибку "use of deleted function ..."
// ----------------------------------------------------------------------------
class TJob {
public:
    TJob() = default;

    // Пользовательский конструктор копирования
    TJob(const TJob &other) {
        JobId = other.JobId;
        BarrierName = other.BarrierName;
        Dependencies = other.Dependencies;
        Children = other.Children;
        RemainDeps.store(other.RemainDeps.load());
        IsFinished.store(other.IsFinished.load());
        IsSuccess.store(other.IsSuccess.load());
    }

    // Оператор присваивания
    TJob& operator=(const TJob &other) {
        if (this != &other) {
            JobId = other.JobId;
            BarrierName = other.BarrierName;
            Dependencies = other.Dependencies;
            Children = other.Children;
            RemainDeps.store(other.RemainDeps.load());
            IsFinished.store(other.IsFinished.load());
            IsSuccess.store(other.IsSuccess.load());
        }
        return *this;
    }

    // Поля
    int JobId = 0;
    std::string BarrierName;
    std::vector<int> Dependencies;
    std::vector<int> Children;

    std::atomic<int> RemainDeps{0};
    std::atomic<bool> IsFinished{false};
    std::atomic<bool> IsSuccess{true};
};

// ----------------------------------------------------------------------------
// «Примитив-барьер» — хранит счётчик оставшихся джоб в группе
// ----------------------------------------------------------------------------
class TBarrierGroup {
public:
    TBarrierGroup() {
        pthread_mutex_init(&Mutex, nullptr);
        pthread_cond_init(&Cond, nullptr);
    }
    ~TBarrierGroup() {
```

```
        pthread_mutex_destroy(&Mutex);
        pthread_cond_destroy(&Cond);
    }

    // Общее число джоб в группе
    int TotalCount = 0;
    // Сколько джоб этой группы ещё не завершились
    int Remaining = 0;

    pthread_mutex_t Mutex;
    pthread_cond_t Cond;
};

// ------------------------------------------------------------------------
// Исполнитель DAG (Directed Acyclic Graph)
// ------------------------------------------------------------------------
class TDagExecutor {
public:
    TDagExecutor();
    ~TDagExecutor();

    // Читаем конфиг
    bool ReadIni(const std::string &filename);

    // Проверяем на наличие только одной компоненты связности
    bool CheckSingleComponent();

    // Проверяем, есть ли цикл
    bool CheckCycle();

    // Проверяем, что есть хотя бы одна стартовая и одна финишная джоба
    bool CheckStartFinish();

    // Запускаем весь DAG
    void RunDAG();

    // --- Дополнительно для тестов ---
    // Дать доступ к карте джоб (только для отладки/тестов)
    const std::map<int, TJob>& GetJobs() const { return Jobs_; }
    int GetMaxParallel() const { return MaxParallel_; }

private:
    bool HasCycleUtil(int jobId,
                      std::map<int, bool> &visited,
                      std::map<int, bool> &recStack);

    static void* JobRunner(void* arg);

    void BarrierArrive(const std::string &barrierName);
    bool IsBarrierUnlocked(const std::string &barrierName);

private:
    // Основные структуры данных:
```

```
    std::map<int, TJob> Jobs_;
    std::map<std::string, TBarrierGroup> BarrierGroups_;
    std::queue<int> ReadyQueue_;

    // Макс. число одновременно исполняемых джоб
    int MaxParallel_ = 2;

    // Глобальный флаг остановки (если какая-то джоба упала)
    std::atomic<bool> StopExecution_{false};

    // Мьютекс и условная переменная для ReadyQueue_
    pthread_mutex_t QueueMutex_;
    pthread_cond_t QueueCond_;

    // Счётчик активных джоб (выполняющихся потоков)
    std::atomic<int> CurrentActiveThreads_{0};
};

#endif
```

dag.cpp

```
#include "dag.h"
#include <sstream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>

pthread_mutex_t g_PrintMutex = PTHREAD_MUTEX_INITIALIZER;

void SafePrint(const std::string &message) {
    pthread_mutex_lock(&g_PrintMutex);
    std::cout << message << std::endl;
    pthread_mutex_unlock(&g_PrintMutex);
}

TDagExecutor::TDagExecutor() {
    srand(time(nullptr));
    pthread_mutex_init(&QueueMutex_, nullptr);
    pthread_cond_init(&QueueCond_, nullptr);
}

TDagExecutor::~TDagExecutor() {
    pthread_mutex_destroy(&QueueMutex_);
    pthread_cond_destroy(&QueueCond_);
}

// ---------------------------------------------------------------------------
// Парсер config.ini
// ---------------------------------------------------------------------------
```

```
bool TDagExecutor::ReadIni(const std::string &filename) {
    std::ifstream fin(filename);
    if (!fin.is_open()) {
        SafePrint("Cannot open config file: " + filename);
        return false;
    }

    // Сбрасываем состояния
    Jobs_.clear();
    BarrierGroups_.clear();
    while (!ReadyQueue_.empty()) {
        ReadyQueue_.pop();
    }
    StopExecution_.store(false);
    CurrentActiveThreads_.store(0);

    std::string line;
    enum class ESection {
        None,
        Jobs,
        Edges,
        Parallel
    };
    ESection currentSection = ESection::None;

    while (std::getline(fin, line)) {
        // Trim
        {
            size_t startPos = 0;
            while (startPos < line.size() && isspace((unsigned char)line[startPos])) {
                startPos++;
            }
            size_t endPos = line.size();
            while (endPos > startPos && isspace((unsigned char)line[endPos-1])) {
                endPos--;
            }
            line = line.substr(startPos, endPos - startPos);
        }

        if (line.empty() || line[0] == '#') {
            continue; // пропускаем комментарии/пустые строки
        }
        if (line == "[Jobs]") {
            currentSection = ESection::Jobs;
            continue;
        } else if (line == "[Edges]") {
            currentSection = ESection::Edges;
            continue;
        } else if (line == "[Parallel]") {
            currentSection = ESection::Parallel;
            continue;
        }
```

```
switch (currentSection) {
case ESection::Jobs: {
    // формат: "job=1 barrier=B1" или "job=4"
    int jobId = 0;
    std::string barrier;

    std::istringstream iss(line);
    std::string token;
    while (iss >> token) {
        size_t pos = token.find('=');
        if (pos != std::string::npos) {
            std::string key = token.substr(0, pos);
            std::string val = token.substr(pos+1);
            if (key == "job") {
                jobId = std::stoi(val);
            } else if (key == "barrier") {
                barrier = val;
            }
        }
    }

    if (jobId <= 0) {
        std::ostringstream oss;
        oss << "Bad job definition line: " << line;
        SafePrint(oss.str());
    } else {
        TJob job;
        job.JobId = jobId;
        job.BarrierName = barrier;
        Jobs_[jobId] = job;
    }
    break;
}
case ESection::Edges: {
    // формат "1->4"
    size_t pos = line.find("->");
    if (pos == std::string::npos) {
        std::ostringstream oss;
        oss << "Bad edge line: " << line;
        SafePrint(oss.str());
        break;
    }
    int from = std::stoi(line.substr(0, pos));
    int to   = std::stoi(line.substr(pos+2));
    if (Jobs_.find(from) == Jobs_.end() || Jobs_.find(to) == Jobs_.end()) {
        std::ostringstream oss;
        oss << "Edge references unknown job: " << line;
        SafePrint(oss.str());
    } else {
        Jobs_[from].Children.push_back(to);
        Jobs_[to].Dependencies.push_back(from);
    }
    break;
```

```
        }
        case ESection::Parallel: {
            MaxParallel_ = std::stoi(line);
            break;
        }
        default:
            // не в секции — пропускаем
            break;
        }
    }
    fin.close();

    // Инициализация полей
    for (auto &kv : Jobs_) {
        kv.second.RemainDeps.store(kv.second.Dependencies.size());
        kv.second.IsFinished.store(false);
        kv.second.IsSuccess.store(true);
    }

    // Инициализация барьеров
    for (auto &kv : Jobs_) {
        const std::string &bn = kv.second.BarrierName;
        if (!bn.empty()) {
            if (BarrierGroups_.find(bn) == BarrierGroups_.end()) {
                TBarrierGroup bg;
                BarrierGroups_[bn] = bg;
            }
            BarrierGroups_[bn].TotalCount++;
            BarrierGroups_[bn].Remaining++;
        }
    }

    return true;
}

// ----------------------------------------------------------------------------
// Проверяем граф на наличие только одной компоненты связности
// ----------------------------------------------------------------------------
bool TDagExecutor::CheckSingleComponent() {
    if (Jobs_.empty()) {
        // Пустой граф — на ваш выбор, считаем "ошибка"
        return false;
    }

    // Построим "неориентированные" связи
    std::map<int, std::vector<int>> undirected;
    for (auto &kv : Jobs_) {
        int id = kv.first;
        undirected[id]; // чтобы существовал пустой вектор
        // Добавим детей (u->v => u--v, v--u)
        for (int child : kv.second.Children) {
            undirected[id].push_back(child);
            undirected[child].push_back(id);
```

```
        }
    }

    // Возьмём первую попавшуюся джобу
    auto it = Jobs_.begin();
    int startId = it->first;

    // DFS/BFS
    std::vector<int> stack;
    stack.push_back(startId);
    std::set<int> visited;
    visited.insert(startId);

    while (!stack.empty()) {
        int curr = stack.back();
        stack.pop_back();
        for (int neigh : undirected[curr]) {
            if (visited.count(neigh) == 0) {
                visited.insert(neigh);
                stack.push_back(neigh);
            }
        }
    }

    // Если посетили все джобы -> одна компонента
    return (visited.size() == Jobs_.size());
}

// ----------------------------------------------------------------------------
// Проверяем граф на наличие цикла (DFS-рекурсивный стек)
// ----------------------------------------------------------------------------
bool TDagExecutor::HasCycleUtil(int jobId,
                                std::map<int, bool> &visited,
                                std::map<int, bool> &recStack)
{
    if (!visited[jobId]) {
        visited[jobId] = true;
        recStack[jobId] = true;

        for (int childId : Jobs_[jobId].Children) {
            if (!visited[childId] && HasCycleUtil(childId, visited, recStack)) {
                return true;
            } else if (recStack[childId]) {
                return true;
            }
        }
    }
    recStack[jobId] = false;
    return false;
}

bool TDagExecutor::CheckCycle() {
    std::map<int, bool> visited;
```

```
    std::map<int, bool> recStack;

    for (auto &kv : Jobs_) {
        visited[kv.first] = false;
        recStack[kv.first] = false;
    }

    for (auto &kv : Jobs_) {
        if (!visited[kv.first]) {
            if (HasCycleUtil(kv.first, visited, recStack)) {
                return true;
            }
        }
    }
    return false;
}

// ---------------------------------------------------------------------------
// Проверка наличия хотя бы одной start-джобы (нет зависимостей)
// и хотя бы одной finish-джобы (нет детей)
// ---------------------------------------------------------------------------
bool TDagExecutor::CheckStartFinish() {
    bool hasStart = false;
    bool hasFinish = false;

    for (auto &kv : Jobs_) {
        if (kv.second.Dependencies.empty()) {
            hasStart = true;
        }
        if (kv.second.Children.empty()) {
            hasFinish = true;
        }
    }
    return (hasStart && hasFinish);
}

// ---------------------------------------------------------------------------
// Проверяем, «разблокирован» ли барьер (Remaining == 0)
// ---------------------------------------------------------------------------
bool TDagExecutor::IsBarrierUnlocked(const std::string &barrierName) {
    if (barrierName.empty()) {
        return true;
    }

    auto it = BarrierGroups_.find(barrierName);
    if (it == BarrierGroups_.end()) {
        // нет такого барьера — считаем, что разблокирован
        return true;
    }

    TBarrierGroup &bg = it->second;
    pthread_mutex_lock(&bg.Mutex);
    bool unlocked = (bg.Remaining == 0);
```

```
    pthread_mutex_unlock(&bg.Mutex);

    return unlocked;
}

// ---------------------------------------------------------------------------
// Уменьшаем счётчик барьера после окончания джобы.
// Если дошли до 0 — барьер разблокирован -> пробуждаем всех
// ---------------------------------------------------------------------------
void TDagExecutor::BarrierArrive(const std::string &barrierName) {
    if (barrierName.empty()) {
        return;
    }

    auto it = BarrierGroups_.find(barrierName);
    if (it == BarrierGroups_.end()) {
        return;
    }

    TBarrierGroup &bg = it->second;

    pthread_mutex_lock(&bg.Mutex);
    bg.Remaining--;
    if (bg.Remaining < 0) {
        bg.Remaining = 0; // safeguard
    }
    if (bg.Remaining == 0) {
        pthread_cond_broadcast(&bg.Cond);
    }
    pthread_mutex_unlock(&bg.Mutex);
}

// ---------------------------------------------------------------------------
// Поток выполнения конкретной джобы
// ---------------------------------------------------------------------------
void* TDagExecutor::JobRunner(void* arg) {
    auto *pairPtr = reinterpret_cast<std::pair<TDagExecutor*, int>*>(arg);
    TDagExecutor *Executor = pairPtr->first;
    int jobId = pairPtr->second;
    delete pairPtr;

    TJob &job = Executor->Jobs_[jobId];

    {
        std::ostringstream oss;
        oss << "[Thread " << pthread_self()
            << "] Starting job " << jobId;
        SafePrint(oss.str());
    }

    // Имитируем работу: sleep от 1 до 3 сек
    bool success = true;
    if (jobId % 2 == 1) {
```

```
    // Случайная имитация ошибки (1 из 5) для нечётных джоб
    int r = rand() % 5;
    if (r == 0) {
        success = false;
    }
}
::sleep(1 + rand() % 3);

if (!success) {
    {
        std::ostringstream oss;
        oss << "!!! Job " << jobId << " FAILED !!!";
        SafePrint(oss.str());
    }
    job.IsSuccess = false;
    Executor->StopExecution_.store(true);
} else {
    job.IsSuccess = true;
    std::ostringstream oss;
    oss << "[Thread " << pthread_self() << "] Finished job "
        << jobId << " SUCCESS";
    SafePrint(oss.str());
}

job.IsFinished = true;

// Сообщаем барьеру
Executor->BarrierArrive(job.BarrierName);

// Если не было глобального стопа и сама джоба успех — «разблокируем» дочерние
if (!Executor->StopExecution_.load() && job.IsSuccess.load()) {
    for (int childId : job.Children) {
        int depsLeft = --Executor->Jobs_[childId].RemainDeps;
        if (depsLeft == 0) {
            const std::string &childBarrier = Executor->Jobs_[childId].BarrierName;
            if (Executor->IsBarrierUnlocked(childBarrier)) {
                pthread_mutex_lock(&Executor->QueueMutex_);
                Executor->ReadyQueue_.push(childId);
                pthread_cond_signal(&Executor->QueueCond_);
                pthread_mutex_unlock(&Executor->QueueMutex_);
            } else {
                std::ostringstream oss;
                oss << "[Thread " << pthread_self()
                    << "] Child job " << childId
                    << " is ready, but waiting for barrier "
                    << childBarrier << " to be unlocked.";
                SafePrint(oss.str());
            }
        }
    }
}

// Освобождаем «слот»
```

```
    Executor->CurrentActiveThreads_.fetch_sub(1);

    pthread_mutex_lock(&Executor->QueueMutex_);
    pthread_cond_broadcast(&Executor->QueueCond_);
    pthread_mutex_unlock(&Executor->QueueMutex_);

    return nullptr;
}

// --------------------------------------------------------------------------
// Основной диспетчер: запускает джобы, контролируя maxParallel
// --------------------------------------------------------------------------
void TDagExecutor::RunDAG() {
    // Добавим в очередь «стартовые» джобы (remainDeps=0) — БЕЗ проверки барьера
    pthread_mutex_lock(&QueueMutex_);
    for (auto &kv : Jobs_) {
        if (kv.second.RemainDeps.load() == 0) {
            ReadyQueue_.push(kv.first);
        }
    }
    pthread_mutex_unlock(&QueueMutex_);

    while (true) {
        pthread_mutex_lock(&QueueMutex_);

        // Используем while для защиты от спонтанных пробуждений
        while (ReadyQueue_.empty() &&
               CurrentActiveThreads_.load() > 0 &&
               !StopExecution_.load())
        {
            pthread_cond_wait(&QueueCond_, &QueueMutex_);
        }

        if (StopExecution_.load()) {
            pthread_mutex_unlock(&QueueMutex_);
            break;
        }

        // Если нет готовых и нет активных — все дела сделаны
        if (ReadyQueue_.empty() && CurrentActiveThreads_.load() == 0) {
            pthread_mutex_unlock(&QueueMutex_);
            break;
        }

        // Пока есть готовые и не превышен лимит — запускаем
        while (!ReadyQueue_.empty() &&
               CurrentActiveThreads_.load() < MaxParallel_ &&
               !StopExecution_.load())
        {
            int jobId = ReadyQueue_.front();
            ReadyQueue_.pop();

            CurrentActiveThreads_.fetch_add(1);
```

```
            // Создаём поток
            pthread_t threadId;
            auto *arg = new std::pair<TDagExecutor*, int>(this, jobId);
            pthread_create(&threadId, nullptr, &TDagExecutor::JobRunner, arg);
            pthread_detach(threadId);
        }

        pthread_mutex_unlock(&QueueMutex_);
    }

    if (StopExecution_.load()) {
        pthread_mutex_lock(&QueueMutex_);
        std::queue<int> emptyQ;
        std::swap(ReadyQueue_, emptyQ);
        pthread_mutex_unlock(&QueueMutex_);

        SafePrint("DAG execution interrupted due to job failure.");
    } else {
        SafePrint("DAG execution completed successfully.");
    }
}
```

CMakeLists.txt

```
add_executable(CP_dag main.cpp src/dag.cpp)

target_include_directories(CP_dag PRIVATE include)
```

main.cpp

```
#include "dag.h"

int main(int argc, char* argv[]) {
    std::string configFile = "config.ini";
    if (argc > 1) {
        configFile = argv[1];
    }

    TDagExecutor Executor;
    if (!Executor.ReadIni(configFile)) {
        return 1;
    }

    if (Executor.CheckCycle()) {
        SafePrint("ERROR: Graph has a cycle!");
        return 2;
    }
    if (!Executor.CheckStartFinish()) {
```

```
        SafePrint("ERROR: Graph has no start/finish jobs!");
        return 3;
    }

    if (!Executor.CheckSingleComponent()) {
        SafePrint("ERROR: Graph has multiple components!");
        return 4;
    }

    Executor.RunDAG();

    return 0;
}
```

CP_test.cpp

```
#include <gtest/gtest.h>
#include <fstream>
#include "dag.h"

bool WriteTestConfig(const std::string &filename, const std::string &content) {
    std::ofstream fout(filename);
    if (!fout.is_open()) return false;
    fout << content;
    fout.close();
    return true;
}

// Проверяем, что парсится нормальный DAG без цикла
TEST(TDagExecutor, ParseWithoutCycle) {
    std::string tempFile = "test_no_cycle.ini";
    // Простой DAG из условия
    std::string content = R"INI(
[Jobs]
job=1 barrier=B1
job=2 barrier=B1
job=3 barrier=B2
job=4
job=5
job=6

[Edges]
1->4
2->4
3->5
4->6
5->6

[Parallel]
4
```

```
)INI";

    ASSERT_TRUE(WriteTestConfig(tempFile, content));

    TDagExecutor Executor;
    bool ok = Executor.ReadIni(tempFile);
    ASSERT_TRUE(ok);

    // Проверяем, что 6 джоб считались
    auto &jobs = Executor.GetJobs();
    EXPECT_EQ(jobs.size(), 6u);

    // Проверяем цикл
    bool cycle = Executor.CheckCycle();
    EXPECT_FALSE(cycle);

    // Проверяем старт/финиш
    bool startFinish = Executor.CheckStartFinish();
    EXPECT_TRUE(startFinish);

    // Проверяем параллель
    EXPECT_EQ(Executor.GetMaxParallel(), 4);
}

// Проверяем, что обнаруживается цикл
TEST(TDagExecutor, ParseWithCycle) {
    std::string tempFile = "test_cycle.ini";
    std::string content = R"INI(
[Jobs]
job=1
job=2
job=3

[Edges]
1->2
2->3
3->2
)INI";

    ASSERT_TRUE(WriteTestConfig(tempFile, content));

    TDagExecutor Executor;
    bool ok = Executor.ReadIni(tempFile);
    ASSERT_TRUE(ok);

    bool cycle = Executor.CheckCycle();
    EXPECT_TRUE(cycle);
}

// Пустой (или некорректный) файл
TEST(TDagExecutor, ParseEmptyConfig) {
    std::string tempFile = "test_empty.ini";
    std::string content = ""; // пустой
```

```
    ASSERT_TRUE(WriteTestConfig(tempFile, content));

    TDagExecutor Executor;
    bool ok = Executor.ReadIni(tempFile);

    ASSERT_TRUE(ok);

    EXPECT_FALSE(Executor.CheckStartFinish());
}

TEST(TDagExecutor, BarrierCheck) {
    std::string content = R"INI(
[Jobs]
job=1 barrier=B1
job=2 barrier=B1
job=3

[Edges]
1->3
2->3

[Parallel]
2
)INI";

    std::string tempFile = "test_barrier.ini";
    ASSERT_TRUE(WriteTestConfig(tempFile, content));

    TDagExecutor Executor;
    bool ok = Executor.ReadIni(tempFile);
    ASSERT_TRUE(ok);

    EXPECT_FALSE(Executor.CheckCycle());
    EXPECT_TRUE(Executor.CheckStartFinish());

    const auto &jobs = Executor.GetJobs();
    ASSERT_EQ(jobs.at(3).Dependencies.size(), 2u);
}

TEST(TDagExecutor, ParallelLimit) {
    std::string content = R"INI(
[Jobs]
job=1
job=2
job=3
job=4

[Edges]
)INI";

    // Тут нет зависимостей, все job "start" => 4 стартовые
    // [Parallel] = 2 => одновременно не более 2
    content += "\n[Parallel]\n2\n";
```

```
    std::string tempFile = "test_parallel.ini";
    ASSERT_TRUE(WriteTestConfig(tempFile, content));

    TDagExecutor Executor;
    bool ok = Executor.ReadIni(tempFile);
    ASSERT_TRUE(ok);

    EXPECT_FALSE(Executor.CheckCycle());
    EXPECT_TRUE(Executor.CheckStartFinish());
    EXPECT_EQ(Executor.GetMaxParallel(), 2);
}

// Проверяем поведение при нескольких компонентах связности
TEST(TDagExecutor, ParseMultipleComponents) {
    // job=1, job=2 связаны, а job=3 изолирована => 2 компоненты
    std::string content = R"INI(
[Jobs]
job=1
job=2
job=3

[Edges]
1->2
[Parallel]
2
)INI";

    std::string tempFile = "test_multicomp.ini";
    ASSERT_TRUE(WriteTestConfig(tempFile, content));

    TDagExecutor Executor;
    bool ok = Executor.ReadIni(tempFile);
    ASSERT_TRUE(ok);

    // Нет цикла
    bool cycle = Executor.CheckCycle();
    EXPECT_FALSE(cycle);

    // start/finish: job=1 (start) => job=2 (finish), job=3 (start & finish, но
изолирован)
    EXPECT_TRUE(Executor.CheckStartFinish());

    bool singleComp = Executor.CheckSingleComponent();
    EXPECT_FALSE(singleComp); // ожидаем false, так как job=3 изолирована
}

int main(int argc, char **argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Выводы

1. **Задача «выполнена»**: мы можем читать DAG из INI, проверять цикл и наличие стартовых/финальных джоб, запускать потоки с ограничением (N) параллельно, останавливать при ошибке, и «держать» зависимые джобы, пока не освободится их барьер.
2. **Самодельный барьер** реализован как «countdown» на каждую группу (по имени). Если Remaining > 0, то зависимые джобы ждут. Когда Remaining == 0 → разблокировка.
3. **Тесты** (Google Test) обеспечивают базовую уверенность, что парсер INI, логика обнаружения цикла, проверка start/finish работают корректно. Мы можем расширять тесты для более детальной проверки барьеров и параллелизма.

В целом, работа над проектом помогла глубже понять фундаментальные концепции параллелизма, синхронизации и DAG (включая подход к организации кода и парсинг конфига)