Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу

«Операционные системы»

Студент: Юрков Евгений Юрьевич
Группа: М8О–212Б–22
Вариант: 37
Преподаватель: Соколов Андрей Алексеевич
Оценка: _____
Дата: _____
Подпись: _____

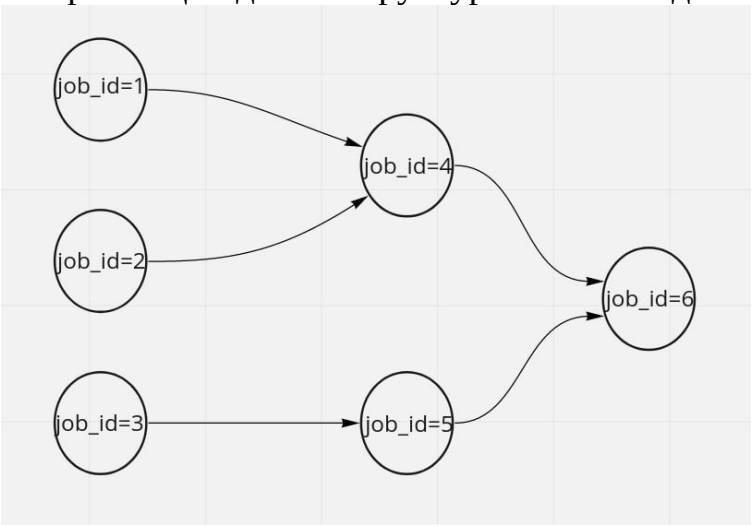
Москва, 2023.

Постановка задачи

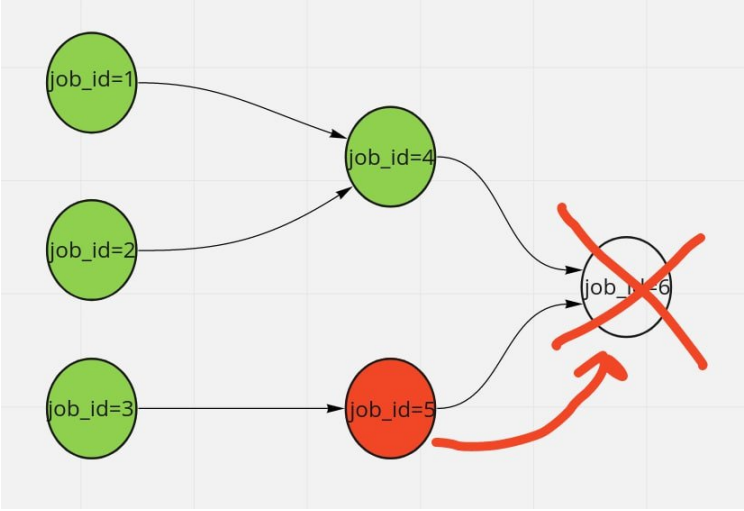
Задание

На языке C\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.

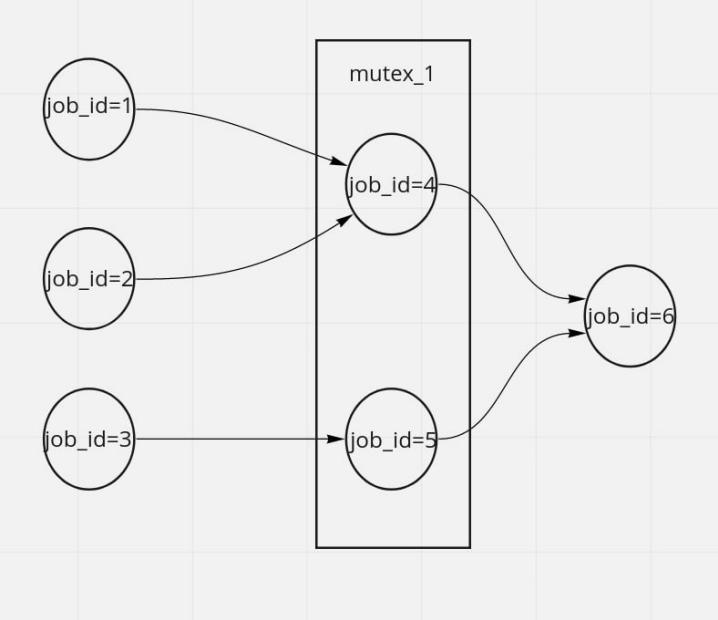


2. При завершении джобы с ошибкой, необходимо прервать выполнение всего DAG’а и всех запущенных джоб.



3. (на оценку 4) Джобы должны запускаться максимально параллельно. Должны быть ограниченны параметром – максимальным числом одновременно выполняемых джоб.
4. (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 выполнятся параллельно с максимальным числом параллельно выполняемых джоб равным степени семафору. Джобы указанные с одним барьером имеют следующие свойство – зависимые от них джобы начнут выполнятся не раньше того момента времени, когда выполнятся все джобы с указанным барьером.



* 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).

Вариант 37: Ini\Mutex

Общие сведения о программе

Основной файл программы - main.cpp. Также используется заголовочные файлы: iostream, thread, mutex, vector, queue, unordered_map, unordered_set, sstream, boost/property_tree/ptree.hpp, boost/property_tree/ini_parser.hpp, wait.h.

В программе используются следующие системные вызовы:

1. **fork** – создание дочернего процесса.
2. **kill** – завершает процесс.
3. **execv** – выполняет файл.
4. **waitpid** – ожидает завершения процесса и принимает от него сигнал

Общий метод и алгоритм решения.

Для обработки ini файла использовалась библиотека boost/property_tree.

Для нахождения циклов используется обход графа в глубину. Текущая вершина помечается 1 и обход продолжается вглубь, когда вершина покидается она помечается 2. Таким образом, если на пути встречается вершина, помеченная 1, то в графе есть цикл.

Для проверки на наличие только одной компоненты связности сначала происходит обход графа и помечаются посещённые вершины. Потом запускается ещё один обход графа, если при нём не посещается ни одна помеченная вершина, то в графе больше одной компоненты связности.

При запуске номера процессов берутся из start_points – туда изначально записаны все номера стартовых джобов. Номера запущенных процессов помещаются в очередь. Для ожидания завершения джоб существует отдельный поток, он берет номера процессов из очереди и ждёт их завершения с помощью waitpid и принимает сигнал, который позволяет понять завершился ли процесс с ошибкой.

Основные файлы программы

main.cpp:

```
#include <vector>
#include <queue>
#include <unordered_map>
#include <unordered_set>
#include <sstream>
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/ini_parser.hpp>
#include <wait.h>

#include <iostream>
#include <thread>
#include <mutex>

using graph = std::vector<std::unordered_set<int>>;

// Создание графа и множества конечных джобов из property tree
graph create_graph(boost::property_tree::ptree& tree,
std::unordered_set<int>& ends) {
    graph res(tree.size());
    for (auto [id, child] : tree) {
        std::stringstream s {child.get<std::string>("next", "")};

        int u = std::stoi(id) - 1;
        if (s.str().empty()) {
            ends.insert(u);
            continue;
        }
```

```
        int v;
        while (s >> v) {
            --v;
            res[v].insert(u);

            if (s.peek() == ',') {
                s.ignore();
            }
        }
    }
    return res;
}
/* поиск цикла в графе
* @return true если нашел, false если не нашел.
*/
bool find_cycle(graph& g, int u, std::vector<int>& visited) {
    if (visited[u] == 1) {
        return true;
    }
    if (visited[u] == 2) {
        return false;
    }
    visited[u] = 1;
    bool res = false;
    for (int v : g[u]) {
        res |= find_cycle(g, v, visited);
    }
    visited[u] = 2;
    return res;
}

// обход дерева в глубину и отметка посещённых вершин
void dfs(graph& g, int u, std::vector<int>& visited) {
    if (visited[u] == 1) {
        return;
    }
    visited[u] = 1;
    bool res = true;
    for (int v : g[u]) {
        dfs(g, v, visited);
    }
}

/* ищет компоненты отделенные от графа
* @return true, если текущий компонент не связан с графом, false, если связь
есть.
*/
bool find_components(graph& g, int u, std::vector<int>& visited) {
    if (visited[u] == 1) {
        return false;
    }
    if (g[u].empty()) {
        return true;
    }
    visited[u] = 1;
    bool res = true;
    for (int v : g[u]) {
        res &= find_components(g, v, visited);
    }
```

```
    return res;
}

// проверка графа на отсутствие циклов
bool check_cycle(graph& g, std::unordered_set<int>& starts) {
    std::vector<int> visited(g.size());
    bool res = true;
    for (int start : starts) {
        res &= !find_cycle(g, start, visited);
    }
    return res;
}
// проверка графа на одну компоненту связности
bool check_connectivity(graph& g, std::unordered_set<int>& starts) {
    std::vector<int> visited(g.size());
    bool res = true;
    dfs(g, *(starts.begin()), visited);
    for (auto itr = std::next(starts.begin(), 1); itr != starts.end(); ++itr)
{
        res &= !find_components(g, *itr, visited);
    }
    return res;
}

/* создание джобы (процесса)
* @param Команда  путь к выполняемому файлу, который необходимо запустить.
* @return pid созданного процесса.
*/
pid_t create_job(std::string command) {
    std::stringstream ss {command};
    std::vector<std::string> strargs;
    std::vector<const char*> args;
    std::string str;
    while (ss >> str) {
        strargs.push_back(str);
        args.push_back(str.c_str());
    }

    pid_t pid = fork();
    if (pid == 0) {
        execv(args[0], const_cast<char * const *>(args.data()));
        pid = -2;
        std::cerr << "exec error" << std::endl;
        exit(-2);
    }

    std::cout << "create " << pid << ": " << command << std::endl;
    return pid;
}
/* убивает все запущенные процессы
* используется в случае ошибки
*/
void killall_jobs(std::queue<std::pair<int, pid_t>>& q) {
    while(!q.empty()) {
        auto [id, pid] = q.front();
        q.pop();
        if (pid != -1)
            kill(pid, SIGINT);
    }
}
```

```
int main(int argc, char** argv) {
    // execl("/bin/sh", "sh", "-c", "ls ../src", nullptr);
    // execl("/bin/sh", "sh", "-c",
"/home/kruyneg/Programming/OOP/build/lab_07", nullptr);
    std::string dagfilename = argv[1];
    // чтение из файла
    boost::property_tree::ptree dag_ptree;
    try {
    boost::property_tree::ini_parser::read_ini(dagfilename/*
"/home/kruyneg/Programming/OS/course_work/src/dag.ini" */, dag_ptree);
    }
    catch (boost::property_tree::ini_parser_error e) {
        std::cerr << e.message() << std::endl;
        return 0;
    }

#ifdef _DEBUG
    for (auto elem : dag_ptree) {
        std::cout << elem.first << std::endl;
        for (auto el : elem.second) {
            std::cout << "  " <<  el.first << std::endl;
            std:: cout << "    " << elem.second.get<std::string>(el.first) <<
std::endl;
            std::cout << std::endl;
        }
    }
#endif

    std::unordered_set<int> end_points, start_points;
    graph g = create_graph(dag_ptree, end_points);

    for (int i = 0; i < g.size(); ++i) {
        if (g[i].empty())
            start_points.insert(i);
    }

    if (start_points.empty()) {
        std::cerr << "Error: Отсутствуют начальные джобы" << std::endl;
        return -1;
    }
    if (end_points.empty()) {
        std::cerr << "Error: Отсутствуют завершающие джобы" << std::endl;
        return -1;
    }

    if (!check_cycle(g, end_points)) {
        std::cerr << "Error: В графе есть циклы" << std::endl;
        return -1;
    }

    if (!check_connectivity(g, end_points)) {
        std::cerr << "Error: В графе больше одной компоненты связности" <<
std::endl;
        return -1;
    }

    // Считываем мьютексы каждой джобы
```

```
    std::unordered_map<std::string, bool> mutex_vals;
    std::unordered_map<int, std::string> mutex_names;

    for (auto [id, node] : dag_ptree) {
        std::string mutex_name = node.get<std::string>("mutex", "");
        if (!mutex_name.empty()) {
            mutex_names[std::stoi(id) - 1] = mutex_name;
            mutex_vals[mutex_name] = true;
        }
    }

    // начинаем запуск DAG'a
    std::queue<std::pair<int, pid_t>> waitq;
    std::mutex qmtx;

    // этот поток ждёт завершения джоб и ловит их ошибки
    std::thread wait_thread([&](){
        while (!end_points.empty() || !waitq.empty()) {
            if (!waitq.empty()) {
                pid_t pid;
                int id;
                {
                    std::lock_guard<std::mutex> lock(qmtx);
                    id = waitq.front().first;
                    pid = waitq.front().second;
                    waitq.pop();
                }
                // проверка нужно ли ждать этот процесс, -1 значит, что
команда была запущена с помощью system()
                if (pid != -1) {
                    int sig;
                    // Ловим ошибку с помощью waitpid
                    std::cout << "wait " << pid << std::endl;
                    waitpid(pid, &sig, 0);
                    if (WIFSIGNALED(sig)) {
                        // если ошибка найдена, то завершаем все процессы и
переходим к завершению программы
                        std::cout << "signal is heared from " << id + 1 <<
std::endl;
                        std::lock_guard<std::mutex> lock(qmtx);
                        end_points = {};
                        killall_jobs(waitq);
                    }
                }
                // Разблокируем мьютекс джобы
                {
                    std::lock_guard<std::mutex> lock(qmtx);
                    mutex_vals[mutex_names[id]] = true;
                }
                // добавляем в очередь запуска готовые к выполнению джобы
                for (int i = 0; i < g.size(); ++i) {
                    if (g[i].count(id)) {
                        g[i].erase(id);
                        if (g[i].empty()) {
                            std::lock_guard<std::mutex> lock(qmtx);
                            start_points.insert(i);
                        }
                    }
                }
                end_points.erase(id);
```

```
            }
        }
    });

    while (!end_points.empty()) {
        // контейнер для запоминания запущенных джобов, чтобы удалить их из
start_points
        std::vector<int> erase_id;
        std::lock_guard<std::mutex> lock(qmtx);
        for (int id : start_points) {
            if ((mutex_names.count(id) &&  mutex_vals[mutex_names[id]])
|| !mutex_names.count(id)) {
                std::string command = dag_ptree.get_child(std::to_string(id +
1)).get<std::string>("command");
                if (command.front() != '.' && command.front() != '/') {
                    // если команда не является выполнимым файлом, то
запускаем её с помощью system()
                    int success = std::system(command.c_str());
                    if (success == -1) {
                        killall_jobs(waitq);
                    }
                    erase_id.push_back(id);
                    waitq.push({id, -1});
                }
                else {
                    // иначе делаем fork exec
                    pid_t pid = create_job(command);

                    // блокируем мьютекс джобы
                    if (mutex_names.count(id))
                        mutex_vals[mutex_names[id]] = false;

                    erase_id.push_back(id);
                    // добавляем процесс в очередь для ожидания
                    waitq.push({id, pid});
                }
            }
        }
        for (int id : erase_id) {
            start_points.erase(id);
        }
    }

    killall_jobs(waitq);

    wait_thread.join();
}
```

Тестирование:

Для тестирования были написаны ещё два файла: errorjob – процесс завершающийся с ошибкой; timerjob – процесс ожидающий 10 секунд и печатающий время начала и конца работы.

timerjob.cpp:

```
#include <iostream>
#include <chrono>
```

```
#include <iomanip>
#include <thread>

using namespace std::chrono_literals;

int main() {
    auto now = std::chrono::system_clock::now();
    std::time_t now_time = std::chrono::system_clock::to_time_t(now);
    std::tm local_tm = *std::localtime(&now_time);
    std::cout << std::put_time(&local_tm, "%H:%M:%S - start\n");

    std::this_thread::sleep_for(10s);

    now = std::chrono::system_clock::now();
    now_time = std::chrono::system_clock::to_time_t(now);
    local_tm = *std::localtime(&now_time);
    std::cout << std::put_time(&local_tm, "%H:%M:%S - end\n");
}
```

errorjob.cpp:

```
#include <vector>

int main() {
    int v[5] = {1, 2, 3, 4, 5};
    v[6] = 1;
    return 0;
}
```

```
[kruyneg@matebook14 build]$ cat ./course_work/tests/error.ini
[1]
command = /home/kruyneg/Programming/OS/build/timerjob
next = 2
[2]
command = /home/kruyneg/Programming/OS/build/errorjob
next = 3, 4
[3]
command = /home/kruyneg/Programming/OS/build/timerjob
[4]
command = /home/kruyneg/Programming/OS/build/timerjob
[kruyneg@matebook14 build]$ ./build/scheduler ./course_work/tests/error.ini
20:01:05 - start
20:01:15 - end
*** stack smashing detected ***: terminated
signal is heared from 2
[kruyneg@matebook14 build]$ cat ./course_work/tests/timers2.ini
[1]
```

```
command = pwd
next = 2
[2]
command = ls
next = 3, 4
[3]
command = /home/kruyneg/Programming/OS/build/timerjob
next = 5
mutex = mt
[4]
command = /home/kruyneg/Programming/OS/build/timerjob
next = 5
mutex = mt
[5]
command = echo DONE!
[kruyneg@matebook14 build]$ ./build/scheduler ./course_work/tests/timers2.ini
/home/kruyneg/Programming/OS
build  course_work  lab1  lab2  lab3  lab4  lab5-7  lab8  output.txt
vgcore.35615
create 3953: /home/kruyneg/Programming/OS/build/timerjob
wait 3953
19:55:29 - start
19:55:39 - end
create 3983: /home/kruyneg/Programming/OS/build/timerjob
wait 3983
19:55:39 - start
19:55:49 - end
DONE!
```

Вывод

Проделав работу, я написал DAG Scheduler. Я реализовал проверку корректности введенного графа и запуск процессов в порядке, установленном в DAG. Также я укрепил свои навыки в проектировании программ, взаимодействующих с процессами.