Московский авиационный институт
(Национальный исследовательский университет)
Факультет "Информационные технологии и прикладная математика"
Кафедра "Вычислительная математика и программирование"

Лабораторная работа №3 по курсу “Операционные системы”

Студент: Живалев Е.А.

Группа: М8О-206Б

Преподаватель: Соколов А.А.

Вариант: 6

Оценка: _____

Дата: _____

Подпись: _____

Москва, 2019

1 Задание

Произвести распараллеленный поиск по ненаправленному графу в ширину. Граф задается набором значений, что хранятся в вершинах, и набором пар связей. Информация по графу хранится в отдельном файле. Необходимо определить есть ли в графе циклы.

В ходе выполнения лабораторной работы были использованы следующие системные вызовы:

- pthread_create - создание нового процесса
- pthread_join - ожидание завершения работы потока
- pthread_exit - завершение потока
- pthread_mutex_lock - захват мьютекса
- pthread_mutex_unlock - освобождение мьютекса

2 Описание работы программы

Я попытался реализовать threadpool, в который я мог бы отправлять задачи(обработку конкретной вершины графа) и получать ответ. Я создал две очереди, одна - для отправки задач, вторая - для получения результатов. Поток, обрабатывая вершину графа, отмечает в массиве used, что вершина была посещена и добавляет в очередь заданий все вершины, соединенные с текущей, а также для них в массиве parent отмечает, что текущая вершина является их родителем. Если же вершина, соединенная с текущей, была посещена, то проверяется, что она является родителем текущей вершины, иначе в графе существует цикл. Операции, связанные с добавлением заданий в очереди, а также обновлением массивов used и parent блокируются во избежании гонки за ресурсами.

3 Исходный код

main.c

```
#include <iostream>
#include <queue>
#include <vector>
#include <pthread.h>
#include <unistd.h>
#include <tbb/concurrent_queue.h>

using Graph_t = std::vector<std::vector<int>>;

struct Work {
    int vertex;
    std::string type;
};

struct Result {
    bool found;
    std::string type;
};

Graph_t graph;
std::vector<bool> used;
std::vector<int> parent;
tbb::concurrent_bounded_queue<Work> tasks;
tbb::concurrent_queue<Result> results;
int numberOfThreads = 1;
int threadsFinished = 0;
pthread_mutex_t lock;
bool found = false;


void* threadPool(void* params) {
    bool done = false;
    Work currentTask;
    while(!done) {
        pthread_mutex_lock(&lock);
        ++threadsFinished;
        int currentlyFinished = threadsFinished;
        pthread_mutex_unlock(&lock);
        if(currentlyFinished + 1 == numberOfThreads) {
            results.push({false, "Done"});
        }
        tasks.pop(currentTask);
        pthread_mutex_lock(&lock);
        --threadsFinished;
        pthread_mutex_unlock(&lock);
        if(currentTask.type == "Done") {
            done = true;
            continue;
        }
        for(const auto& vertex : graph[currentTask.vertex]) {
            pthread_mutex_lock(&lock);
            if(!used[vertex]) {
                used[vertex] = true;
                parent[vertex] = currentTask.vertex;
                tasks.push({vertex,  "Continue"});
            } else if(parent[currentTask.vertex] != vertex) {
```

```
                found = true;
                pthread_mutex_unlock(&lock);
                continue;
            }
            pthread_mutex_unlock(&lock);
        }
        results.push({false, "Continue"});
    }
    pthread_exit(NULL);
}


void multithreadedBFS(int startingVertex) {
    pthread_t threads[numberOfThreads];
    for(int i = 0; i < numberOfThreads; ++i) {
        pthread_create(&threads[i], NULL, threadPool, (void*)0);
    }
    tasks.push({startingVertex, "Continue"});
    bool done = false;
    while(!done) {
        Result r;
        results.try_pop(r);
        if(r.type == "Done") {
            done = true;
        }
    }
    for(int i = 0; i < numberOfThreads; ++i) {
        tasks.push({startingVertex, "Done"});
    }
    for(int i = 0; i < numberOfThreads; ++i) {
        pthread_join(threads[i], NULL);
    }
    std::cout << "Mutlithreaded: ";
    if(found) {
        std::cout << "Cycle found" << std::endl;
    } else {
        std::cout << "Cycle not found" << std::endl;
    }
}

bool singlethreadedBFS(const Graph_t& graph, int startingVertex) {
    std::queue<int> q;
    q.push(startingVertex);
    std::vector<bool> used(graph.size());
    std::vector<int> parent(graph.size(), -1);
    used[startingVertex] = true;
    while(!q.empty()) {
        int currentVertex = q.front();
        q.pop();
        for(const auto& v : graph[currentVertex]) {
            if(!used[v]) {
                used[v] = true;
                q.push(v);
                parent[v] = currentVertex;
            } else if (parent[currentVertex] != v) {
                return true;
            }
        }
    }
```

```
116     return false;
117 }
118
119 int main() {
120     int n, m;
121     std::cin >> n >> m;
122     graph.resize(n);
123     for(int i = 0; i < m; ++i) {
124         int u, v;
125         std::cin >> u >> v;
126         u -= 1;
127         v -= 1;
128         graph[u].push_back(v);
129         graph[v].push_back(u);
130     }
131     int s;
132     std::cin >> s;
133     if(s < 0 || s > n) {
134         std::cout << "Node with such ID doesn't exist" << std::
    endl;
135         return -1;
136     }
137     s -= 1;
138     std::cout << "Singlethreaded:";
139     if(singlethreadedBFS(graph, s)) {
140         std::cout << "Cycle found" << std::endl;
141     } else {
142         std::cout << "Cycle not found" << std::endl;
143     }
144     std::cin >> numberOfThreads;
145     if(numberOfThreads < 1) {
146         std::cout << "Number of threads must be at least 1" << std
    ::endl;
147         return -1;
148     }
149     if(numberOfThreads > n - 1) {
150         std::cout << "Number of threads can't be greater than
    number of nodes - 1" << std::endl;
151         return -1;
152     }
153     std::vector<pthread_t> threads(numberOfThreads);
154     used.assign(graph.size(), false);
155     parent.assign(graph.size(), -1);
156     pthread_mutex_init(&lock, NULL);
157     multithreadedBFS(s);
158     pthread_mutex_destroy(&lock);
159     return 0;
160 }
```

4 Консоль

```
qelderdelta@qelderdelta-UX331UA:~/Study/OS/os_lab_3/src/build$ ./lab3
3 3
1 2
2 3
3 1
1
Singlethreaded: Cycle found
2
Multithreaded: Cycle found
qelderdelta@qelderdelta-UX331UA:~/Study/OS/os_lab_3/src/build$ ./lab3
3 2
1 2
2 3
1
Singlethreaded: Cycle not found
2
Multithreaded: Cycle not found
```

5 Выводы

В ходе выполнения лабораторной работы я получил очень важный почти для любого разработчика опыт написания многопоточных программ. Конкретно я познакомился с тем, как устроены потоки в ОС Linux, а также с устройством такого примитива синхронизации как мьютекс.