НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра параллельных вычислительных технологий

ОТЧЕТ ПРИНЯТ

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_ / \_\_\_\_\_\_\_\_\_\_\_ /

(подпись) (расшифровка)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_ г.

**ОТЧЕТ**

по лабораторной работе № 2

Программирование взаимодействующих потоков

по дисциплине Параллельное программирование

Студенты гр. ПМИ-02

Дюков Богдан Витальевич,

Сидоров Даниил Игоревич

Новосибирск-2024

1. **Цель работы**

Познакомиться со средствами планирования и синхронизации потоков; научиться работать с потоками, которые обмениваются информацией между собой.

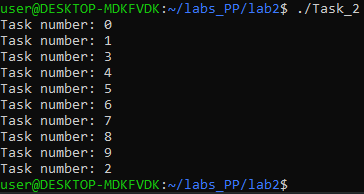
1. **Ход работы**

1. Изучили теоретическую часть из раздела 2 и 3 настоящего пособия.

2. Записали программу из примера 4. Скомпилировали ее и проверили корректность

работы.

Перед компиляцией добавили в функцию do\_task() вывод номера задания:



3. Добавили в пример 4 вывод информации о том, какой поток какую задачу взял на

исполнение, а также некоторое достаточно продолжительное вычисление в функцию do\_task().

В качестве продолжительного вычисления в функции do\_task() выступает приостановка выполнения потока с помощью sleep(). Также добавили в поточную функцию случайную задержку, что приведет к большей вероятности появления “гонки” между потоками.

Текст программы:

#include <cstdlib>

#include <iostream>

#include <cstring>

#include <pthread.h>

#include <sstream>

#include <unistd.h>

using namespace std;

#define err\_exit(code, str) { cerr << str << ": " << strerror(code) << endl; exit(EXIT\_FAILURE); }

const int TASKS\_COUNT = 10;

int task\_list[TASKS\_COUNT]; // Массив заданий

int current\_task = 0; // Указатель на текущее задание

// Массив выполненных заданий

bool tasks\_done\_by\_thread1[TASKS\_COUNT] = {false};

bool tasks\_done\_by\_thread2[TASKS\_COUNT] = {false};

// Мьютекс

pthread\_mutex\_t mutex;

// Функция, выполняющая продолжительную операцию

void do\_task(int task\_no)

{

    sleep(2);

}

// Функция, выполняемая потоком

void \*thread\_job(void \*arg)

{

int task\_no;

int status;

int thread\_id = \*(int \*)arg;

// Перебираем в цикле доступные задания

while(true)

{

// Захватываем мьютекс для исключительного доступа

// к указателю текущего задания (переменная current\_task)

status = pthread\_mutex\_lock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot lock mutex");

}

// Запоминаем номер текущего задания, которое будем исполнять

task\_no = current\_task;

sleep(rand() % 2 + 1);

// Сдвигаем указатель текущего задания на следующее

current\_task++;

// Освобождаем мьютекс

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

// Если запомненный номер задания не превышает

// количества заданий, вызываем функцию, которая

// выполнит задание.

// В противном случае завершаем работу потока

if(task\_no < TASKS\_COUNT)

         {

             do\_task(task\_no);

             if(thread\_id == 1)

{

tasks\_done\_by\_thread1[task\_no] = true;

}

             else

{

tasks\_done\_by\_thread2[task\_no] = true;

}

         }

         else

         {

         return NULL;

       }

}

}

int main()

{

srand(time(0));

// Идентификаторы потоков

pthread\_t thread1;

pthread\_t thread2;

int thread1\_id = 1;

     int thread2\_id = 2;

int status; // Код ошибки

// Инициализируем массив заданий случайными числами

for(int i=0; i < TASKS\_COUNT; ++i)

{

task\_list[i] = rand() % TASKS\_COUNT;

}

// Инициализируем мьютекс

status = pthread\_mutex\_init(&mutex, NULL);

if(status != 0)

{

err\_exit(status, "Cannot initialize mutex");

}

// Создаем потоки

status = pthread\_create(&thread1, NULL, thread\_job, &thread1\_id);

if(status != 0)

{

err\_exit(status, "Cannot create thread 1");

}

status = pthread\_create(&thread2, NULL, thread\_job, &thread2\_id);

if(status != 0)

{

err\_exit(status, "Cannot create thread 2");

}

status = pthread\_join(thread1, NULL);

if(status != 0)

{

err\_exit(status, "Cannot join a thread 1");

}

status = pthread\_join(thread2, NULL);

if(status != 0)

{

err\_exit(status, "Cannot join a thread 2");

}

// Выводим, какие задания были выполнены каждым потоком

     cout << "Tasks done by thread 1: ";

     for(int i = 0; i < TASKS\_COUNT; i++)

     {

if(tasks\_done\_by\_thread1[i])

{

cout << i << " ";

}

     }

     cout << endl;

     cout << "Tasks done by thread 2: ";

     for(int i = 0; i < TASKS\_COUNT; i++)

     {

       if(tasks\_done\_by\_thread2[i])

{

cout << i << " ";

}

     }

     cout << endl;

// Освобождаем ресурсы, связанные с мьютексом

pthread\_mutex\_destroy(&mutex);

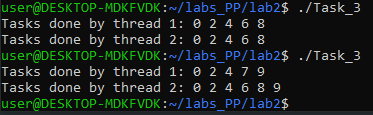
return 0;

}

Проверим, что при использовании мьютекса все задачи выполняются, причем каждая – не более одного раза:



Отключим вызовы функций pthread\_mutex\_lock() и pthread\_mutex\_unlock() и убедимся, что при отсутствии мьютекса некоторые задачи будут выполняться дважды, тогда как другие не выполнятся ни разу:



На примере убедились, что использование мьютекса необходимо для гарантии корректности работы в многопоточной среде.

4. Сравнили скорости работы примитивов синхронизации – мьютекса и спинлока.

Выяснили, как количество потоков, одновременно обращающихся к ресурсу, защищенному примитивом синхронизации, влияет на скорость исполнения программы.

Для этого была реализована программа, в которой производятся различные замеры: замеры времени захвата и освобождения мьютекса и спинлока, а также замеры времени работы двух поточных функций, выполняющих сказанные ранее действия. Время измеряется в секундах.

Текст программы:

#include <iostream>

#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>

#include <cstdlib>

#include <cstring>

#include <iomanip>

#include "../timer.h"

using namespace std;

#define err\_exit(code, str) { cerr << str << ": " << strerror(code) << endl; exit(EXIT\_FAILURE); }

const int NUM\_OF\_THREADS\_DEFAULT = 10;

pthread\_mutex\_t mutex;

pthread\_spinlock\_t spinlock;

// Скорость работы мьютекса

double get\_mutex\_working\_time()

{

int status;

Timer time;

status = pthread\_mutex\_lock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot lock mutex");

}

// Критическая секция

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

return time.elapsed();

}

// Скорость работы спинлока

double get\_spinlock\_working\_time()

{

int status;

Timer time;

status = pthread\_spin\_lock(&spinlock);

if(status != 0)

{

err\_exit(status, "Cannot lock spinlock");

}

// Критическая секция

status = pthread\_spin\_unlock(&spinlock);

if(status != 0)

{

err\_exit(status, "Cannot unlock spinlock");

}

return time.elapsed();

}

void \*mutex\_test(void \*arg)

{

int status = pthread\_mutex\_lock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot lock mutex");

}

// Критическая секция

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

     return NULL;

}

void \*spinlock\_test(void \*arg)

{

int status = pthread\_spin\_lock(&spinlock);

if(status != 0)

{

err\_exit(status, "Cannot lock spinlock");

}

// Критическая секция

status = pthread\_spin\_unlock(&spinlock);

if(status != 0)

{

err\_exit(status, "Cannot unlock spinlock");

}

     return NULL;

}

int main(int argc, char \*argv[])

{

int num\_of\_threads = (argc > 1 ? stoi(argv[1]) : NUM\_OF\_THREADS\_DEFAULT);

     pthread\_t threads[num\_of\_threads];

     int status;

     status = pthread\_mutex\_init(&mutex, NULL);

     if(status != 0)

{

         err\_exit(status, "Cannot initialize mutex");

     }

     status = pthread\_spin\_init(&spinlock, PTHREAD\_PROCESS\_PRIVATE);

     if(status != 0)

{

         err\_exit(status, "Cannot initialize spinlock");

    }

cout << "Mutex test time:     " << fixed << setprecision(10) << get\_mutex\_working\_time() << endl;

cout << "Spinlock test time:  " << std::fixed << std::setprecision(10) << get\_spinlock\_working\_time() << endl << endl;

     // Тест мьютекса

     Timer timer1;

     for(int i = 0; i < num\_of\_threads; i++)

{

         status = pthread\_create(&threads[i], NULL, mutex\_test, NULL);

         if(status != 0)

{

             err\_exit(status, "Cannot create thread for mutex test");

         }

     }

     for(int i = 0; i < num\_of\_threads; i++)

{

         status = pthread\_join(threads[i], NULL);

if(status != 0)

{

err\_exit(status, "Cannot join a thread");

}

     }

     cout << "Mutex test time multithread:     " << fixed << setprecision(8) << timer1.elapsed() << endl;

     // Тест спинлока

     Timer timer2;

     for(int i = 0; i < num\_of\_threads; i++)

{

         status = pthread\_create(&threads[i], NULL, spinlock\_test, NULL);

         if(status != 0)

{

             err\_exit(status, "Cannot create thread for spinlock test");

         }

     }

     for(int i = 0; i < num\_of\_threads; i++)

{

         pthread\_join(threads[i], NULL);

if(status != 0)

{

err\_exit(status, "Cannot join a thread");

}

     }

     cout << "Spinlock test time multithread:  " << std::fixed << std::setprecision(8) << timer2.elapsed() << endl;

     // Освобождаем ресурсы

     pthread\_mutex\_destroy(&mutex);

     pthread\_spin\_destroy(&spinlock);

     return 0;

}

Сравним скорости работы примитивов синхронизации – мьютекса и спинлока:



В данном случае спинлок работает быстрее, чем мьютекс, поскольку ожидание блокировки короткое (он активно занимает процессорное время, в том время как мьютекс переводят ожидающий поток в неактивное состояние).

Замеры времени для разного количества потоков:

| Количество потоков | Мьютекс | Спинлок |
| --- | --- | --- |
| 50 | 0.00170666 | 0.00144215 |
| 100 | 0.00299627 | 0.00186372 |
| 500 | 0.01115549 | 0.01055709 |
| 1000 | 0.02125729 | 0.02099459 |
| 5000 | 0.10981935 | 0.10365083 |
| 10000 | 0.21286748 | 0.20977508 |

Спинлоки в целом работают быстрее, чем мьютексы, независимо от количества потоков. Однако стоит отметить, что разница во времени работы между мьютексами и спинлоками уменьшается с увеличением количества потоков. Это связано с тем, что при большом количестве потоков вероятность долгого ожидания блокировки увеличивается, и в этом случае преимущества спинлоков становятся менее выраженными.

5. Реализовали условную переменную с помощью мьютекса и цикла ожидания.

Для этого реализуем простую модель производителя-потребителя с использованием мьютекса и цикла ожидания вместо условной переменной. Производитель создает товары, а потребитель их потребляет. Код контролирует этот процесс следующим образом:

1. **Производитель** захватывает мьютекс и проверяет, готов ли товар к потреблению. Если товар готов, производитель освобождает мьютекс и уступает процессорное время другим потокам, пока потребитель не потребует товар. Как только товар потреблен, производитель производит новый товар, указывая с помощью флага, что товар произведен, и уменьшает количество товаров, которые осталось произвести.
2. **Потребитель** захватывает мьютекс и проверяет, готов ли товар к потреблению. Если товар не готов, потребитель освобождает мьютекс и уступает процессорное время другим потокам, пока производитель не произведет новый товар. Как только товар готов, потребитель потребляет его и указывает с помощью флага, что товар потреблен.

Текст программы:

#include <iostream>

#include <pthread.h>

#include <cstdlib>

#include <cstring>

#include <unistd.h>

#define err\_exit(code, str) { cerr << str << ": " << strerror(code) << endl; exit(EXIT\_FAILURE); }

#define NUM\_PRODUCTS 7

using namespace std;

pthread\_mutex\_t mutex;

// Количество товаров, которые осталось произвести и потребить

int products\_count = NUM\_PRODUCTS;

// Флаг, указывающий, готов ли продукт

bool product\_ready = false;

// Функция, выполняемая потоком-производителем

void \*producer(void \*arg)

{

int status;

while(true)

{

// Захватываем мьютекс

status = pthread\_mutex\_lock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot lock mutex");

}

// Ждем, пока потребитель не потребует товар или не завершено производство

while(product\_ready && products\_count > 0)

{

// Освобождаем мьютекс, чтобы другой поток мог его захватить

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

// Уступаем процессорное время другим потокам

sched\_yield();

// Снова захватываем мьютекс

status = pthread\_mutex\_lock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot lock mutex");

}

}

// Не допускаем создание товара, если производство завершено

if(products\_count > 0)

{

// Производим товар

cout << "Produced product №" << NUM\_PRODUCTS - products\_count + 1 << endl;

product\_ready = true;

// Уменьшаем количество товаров, которые осталось произвести

products\_count--;

}

// Обеспечиваем своевременное завершение потока-производителя

if(products\_count == 0)

{

pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

return NULL;

}

// Освобождаем мьютекс

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

}

}

// Функция, выполняемая потоком-потребителем

void \*consumer(void \*arg)

{

int status;

while(true)

{

status = pthread\_mutex\_lock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot lock mutex");

}

// Ждем, пока производитель не произведет товар или не завершено производство

while(!product\_ready && products\_count > 0)

{

// Освобождаем мьютекс, чтобы другой поток мог его захватить

pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

// Уступаем процессорное время другим потокам

sched\_yield();

// Снова захватываем мьютекс

pthread\_mutex\_lock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot lock mutex");

}

}

// Потребляем товар, если он готов, иначе производство завершено и поток потребителя завершается

if(product\_ready)

{

// Потребляем товар

cout << "Consumed product №" << NUM\_PRODUCTS - products\_count << endl << endl;

// Указываем, что товар потреблен

product\_ready = false;

}

else

{

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

return NULL;

}

// Освобождаем мьютекс

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

}

}

int main()

{

    pthread\_t thread1, thread2;

     int status;

     // Инициализируем мьютекс

     status = pthread\_mutex\_init(&mutex, NULL);

     if(status != 0)

{

         err\_exit(status, "Cannot initialize mutex");

     }

     // Создаем потоки

     status = pthread\_create(&thread1, NULL, producer, NULL);

     if(status != 0)

{

         err\_exit(status, "Cannot create producer thread");

     }

     status = pthread\_create(&thread2, NULL, consumer, NULL);

     if(status != 0)

{

         err\_exit(status, "Cannot create consumer thread");

     }

     // Дожидаемся завершения потоков

     status = pthread\_join(thread1, NULL);

if(status != 0)

{

err\_exit(status, "Cannot join a producer thread");

}

     status = pthread\_join(thread2, NULL);

if(status != 0)

{

err\_exit(status, "Cannot join a consumer thread");

}

     // Освобождаем ресурсы, связанные с мьютексом

     pthread\_mutex\_destroy(&mutex);

     return 0;

}

Результаты работы программы:



6. Реализовали функцию для выполнения вычислений по модели MapReduce.

С помощью MapReduce реализуется подсчет количества гласных букв в наборе строк. В качестве параметров функция принимает набор строк для обработки, имена функций map и reduce и количество разрешенных потоков.

В качестве реализации параллельного применения функции map к каждому элементу массива использовали результаты пункта 8 порядка выполнения лабораторной работы № 1. Каждый поток в функции map подсчитывает количество гласных букв в своем сегменте строк.

Также реализовали параллельное применение функции reduce к словарю, полученному в результате работы функции map. Каждый поток в функции reduce суммирует значения по ключу из словаря и агрегирует их в результирующий словарь.

Текст программы:

#include <map>

#include <string>

#include <algorithm>

#include <cctype>

#include <vector>

#include <pthread.h>

#include <iostream>

#include <cstring>

#include "../timer.h"

#define err\_exit(code, str) { cerr << str << ": " << strerror(code) << endl; exit(EXIT\_FAILURE); }

using namespace std;

const int NUM\_OF\_THREADS\_DEFAULT = 4;

const int NUM\_OF\_LINES\_DEFAULT = 2;

const int AUTO\_DUPLICATION\_DEFAULT = false;

// Строка гласных и строка по умолчанию

const string vowels = "aeiou";

const string LINE\_DEFAULT = "Hello World! How do you like this program?";

pthread\_mutex\_t mutex;

// Структура для передачи параметров в поток

struct MapThreadParams

{

     vector<string>\* lines;  // Указатель на вектор строк

int begin; // Начало диапазона

     int end; // Конец диапазона

vector<map<char, int>>\* map\_results; // Указатель на вектор результатов

};

// Структура для передачи параметров в поток

struct ReduceThreadParams

{

     vector<map<char, int>>\* map\_results; // Указатель на вектор результатов

int begin; // Начало диапазона

     int end; // Конец диапазона

map<char, int>\* reduce\_result; // Указатель на глобальный результат

};

// Функция, выполняемая потоком

// Подсчитывает количество гласных в сегменте строк

void\* map\_thread(void\* arg)

{

     MapThreadParams\* params = (MapThreadParams\*)arg;

     for (int i = params->begin; i <= params->end; i++)

{

         for (char& symbol : (\*params->lines)[i])

         {

auto lower\_symbol = tolower(symbol);

             if (vowels.find(lower\_symbol) != string::npos)

             {

                 (\*params->map\_results)[i][lower\_symbol]++;

             }

         }

     }

     return NULL;

}

// Функция, выполняемая потоком

// Суммирует значения по ключу из map\_results

void\* reduce\_thread(void\* arg)

{

     ReduceThreadParams\* params = (ReduceThreadParams\*)arg;

int status;

     for (int i = params->begin; i <= params->end; i++)

{

        for (const auto& pair : (\*params->map\_results)[i])

         {

             status = pthread\_mutex\_lock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot lock mutex");

}

             (\*params->reduce\_result)[pair.first] += pair.second;

             status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

         }

     }

     return NULL;

}

// Функция, выполняющая вычисления по модели MapReduce

map<char, int> map\_reduce(vector<string>& lines,

  void\* (\*map\_thread)(void\*),

  void\* (\*reduce\_thread)(void\*),

  int num\_of\_threads)

{

// Параллельное применение map

// Отбрасываем лишние потоки (если необходимо)

     int map\_num\_of\_threads = num\_of\_threads > lines.size() ? lines.size() : num\_of\_threads;

     vector<map<char, int>> map\_result(lines.size());

     vector<pthread\_t> threads(map\_num\_of\_threads);

     vector<MapThreadParams> threadParams(map\_num\_of\_threads);

int status;

// Вычисление целого и остатка

int quotient = lines.size() / map\_num\_of\_threads;

int remainder = lines.size() % map\_num\_of\_threads;

// Создаем потоки, каждый поток обрабатывает свой сегмент строк

     for (int i = 0; i < map\_num\_of\_threads; ++i)

{

         int begin = i \* quotient + min(i, remainder);

int end = begin + quotient - (i < remainder ? 0 : 1);

         threadParams[i] = {&lines, begin, end, &map\_result};

         status = pthread\_create(&threads[i], NULL, map\_thread, &threadParams[i]);

if(status != 0)

{

             err\_exit(status, "Cannot create a thread");

         }

     }

// Ожидаем завершения map-потоков

     for (auto& thread : threads)

{

         status = pthread\_join(thread, NULL);

if(status != 0)

{

             err\_exit(status, "Cannot join a thread");

         }

     }

// Параллельное применение reduce

// Отбрасываем лишние потоки (если необходимо)

     int reduce\_num\_of\_threads = num\_of\_threads > map\_result.size() ? map\_result.size() : num\_of\_threads;

     vector<pthread\_t> reduce\_threads(reduce\_num\_of\_threads);

     vector<ReduceThreadParams> reduceThreadParams(reduce\_num\_of\_threads);

map<char, int> reduce\_result = {{vowels[0], 0}, {vowels[1], 0}, {vowels[2], 0}, {vowels[3], 0}, {vowels[4], 0}};

// Вычисление целого и остатка

     quotient = map\_result.size() / reduce\_num\_of\_threads;

remainder = map\_result.size() % reduce\_num\_of\_threads;

    // Создаем потоки, каждый поток обрабатывает свой сегмент результатов

     for (int i = 0; i < reduce\_num\_of\_threads; ++i)

{

         int begin = i \* quotient + min(i, remainder);

int end = begin + quotient - (i < remainder ? 0 : 1);

         reduceThreadParams[i] = {&map\_result, begin, end, &reduce\_result};

         status = pthread\_create(&reduce\_threads[i], NULL, reduce\_thread, &reduceThreadParams[i]);

if(status != 0)

{

             err\_exit(status, "Cannot create a thread");

         }

     }

     // Ожидаем завершения reduce-потоков

     for (auto& thread : reduce\_threads)

{

         pthread\_join(thread, NULL);

if(status != 0)

{

             err\_exit(status, "Cannot join a thread");

         }

     }

return reduce\_result;

}

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

{

int num\_of\_threads = (argc > 1 ? stoi(argv[1]) : NUM\_OF\_THREADS\_DEFAULT);

int num\_of\_lines = (argc > 2 ? stoi(argv[2]) : NUM\_OF\_LINES\_DEFAULT);

bool auto\_duplication = (argc > 3 && (stoi(argv[3]) == 0 || stoi(argv[3]) == 1) ? stoi(argv[3]) : AUTO\_DUPLICATION\_DEFAULT);

// Исходная коллекция строк

vector<string> lines;

// Пользователь или решает сам ввести набор строк

// Или выбирает набор по умолчанию (введя num\_of\_lines = 0)

if(!auto\_duplication)

{

string line;

cout << "Enter " << num\_of\_lines << " lines: " << endl;

cout << "-----------------------------------------------------" << endl;

for(int i = 0; i < num\_of\_lines; i++)

{

getline(cin, line);

lines.push\_back(line);

}

cout << "-----------------------------------------------------" << endl;

}

else

{

cout << "The set contains " << to\_string(num\_of\_lines) << " lines of the form: " << LINE\_DEFAULT << endl;

for(int i = 0; i < num\_of\_lines; i++)

{

lines.push\_back(LINE\_DEFAULT);

}

}

// Инициализируем мьютекс

int status = pthread\_mutex\_init(&mutex, NULL);

     if(status != 0)

{

         err\_exit(status, "Cannot initialize mutex");

     }

Timer time;

    // Вызываем функцию MapReduce

     auto result = map\_reduce(lines, map\_thread, reduce\_thread, num\_of\_threads);

cout << endl << "MapReduce execution time: " << time.elapsed() << endl << endl;

cout << "The number of vowels in a set of strings: " << endl;

     // Выводим результат

     for (const auto& pair : result)

     {

         cout << pair.first << " = " << pair.second << "\n";

     }

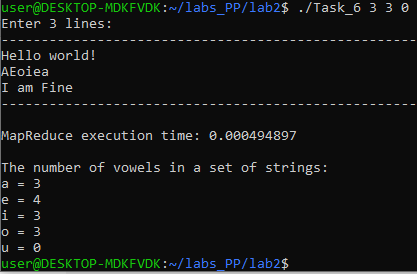
// Освобождаем ресурсы

pthread\_mutex\_destroy(&mutex);

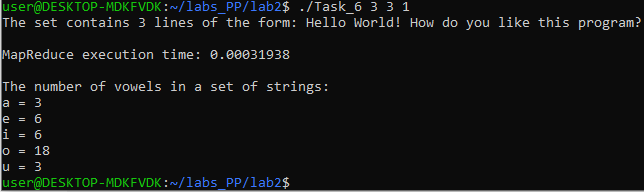
     return 0;

}

Пример работы программы без автозаполнения (3 потока, 3 строки в наборе и самостоятельное заполнение набора):



Пример с автозаполнением:



Оценим скорость работы модели MapReduce:

| Число потоков | Число строк в наборе | | | | |
| --- | --- | --- | --- | --- | --- |
| 10 | 100 | 1000 | 10000 | 100000 |
| 2 | 0.0002985 | 0.0008634 | 0.0024431 | 0.0179668 | 0.1730544 |
| 4 | 0.0004357 | 0.0005459 | 0.0016898 | 0.0147112 | 0.1436648 |
| 8 | 0.0009711 | 0.0015367 | 0.0027727 | 0.0160162 | 0.1768512 |
| 16 | 0.0010039 | 0.0018002 | 0.0036779 | 0.0283682 | 0.2872562 |

Увеличение числа потоков до 4 приводит к уменьшению времени выполнения, что указывает на эффективность параллельной обработки. Однако дальнейшее увеличение числа потоков оказывается бесполезным.

7. Реализовали задачу производитель-потребитель с несколькими потоками-потребителями, одним потоком-производителем и несколькими местами на складе. За основу взяли программу из примера 7.

Пул потоков в нашей реализации представлен в виде массива, который включает в себя поток-производителя и множество потоков-потребителей, количество которых определяется пользователем. Хранилище товаров представлено в виде очереди.

**Алгоритм работы потока-производителя:**

1. Производитель начинает свою работу, его цель - произвести определенное количество товаров.
2. Производитель пытается заблокировать мьютекс для безопасного доступа к хранилищу, после чего создает товар.
3. Если хранилище полно, производитель ожидает, пока потребитель не освободит место в хранилище.
4. Как только в хранилище появляется свободное место, производитель помещает в него новый товар и увеличивает счетчик произведенных товаров.
5. Производитель посылает сигнал о том, что в хранилище появился новый товар, и разблокирует мьютекс.
6. Процесс повторяется до тех пор, пока производитель не произведет целевое число товаров.

**Алгоритм работы потока-потребителя:**

1. Потребитель начинает свою работу.
2. Потребитель пытается заблокировать мьютекс для безопасного доступа к хранилищу.
3. Если хранилище пусто, потребитель ожидает, пока производитель не добавит в него новый товар.
4. Как только в хранилище появляется товар, потребитель забирает его и разблокирует мьютекс.
5. Потребитель посылает сигнал о том, что он забрал товар из хранилища.
6. Если все товары произведены и хранилище пусто, потребитель завершает свою работу. В противном случае процесс повторяется.

В нашей реализации используются мьютексы и условные переменные для синхронизации работы потоков и безопасного доступа к общему ресурсу (хранилищу).

Текст программы:

#include <cstdlib>

#include <cstring>

#include <iostream>

#include <unistd.h>

#include <pthread.h>

#include <queue>

#include <sstream>

#include <vector>

#define err\_exit(code, str) { cerr << str << ": " << strerror(code) << endl; exit(EXIT\_FAILURE); }

#define NUM\_OF\_SLOTS 4 // Емкость хранилища

#define CAPACITY 7   // Общее число товаров, которое должен произвести производитель, прежде чем завершить работу

using namespace std;

const int NUM\_OF\_CONSUMERS\_DEFAULT = 2;

const int CONSUMER\_TIME\_DEFAULT = 2;

const int PRODUCER\_TIME\_DEFAULT = 2;

// Структура для передачи параметров в поток-производитель/потребитель

struct ThreadParams

{

     int index; // Индекс потока

int time; // Время ожидания перед очередным проходом алгоритма производителя/потребителя

};

// Хранилище

queue<int> warehouse;

// Мьютекс и условная переменная для корректной работы с хранилищем

pthread\_mutex\_t mutex;

pthread\_cond\_t cond;

// Счетчик произведенных продуктов

int number\_of\_product\_produced = 0;

// Функция для корректного вывода

void print\_data(string message)

{

ostringstream out;

out << message;

cout << out.str();

}

// Функция, выполняемая потоком-производителем

void \*producer(void \*arg)

{

int producer\_time = static\_cast<ThreadParams\*>(arg)->time;

    int status;

int product;

print\_data("Поток Производителя начинает свою работу! Цель: произвести " + to\_string(CAPACITY) + " товаров.\n\n");

// Выполняем производство до определенного числа произведенных продуктов

     while(true)

     {

// Задержка перед очередным производством

sleep(producer\_time);

// Пытаемся получить доступ к общему ресурсу

         status = pthread\_mutex\_lock(&mutex);

if(status != 0)

         {

             err\_exit(status, "Cannot lock mutex");

        }

// Допускаем создание товара только если производство не завершено

if(number\_of\_product\_produced < CAPACITY)

{

product = rand() % 50 + 1;

print\_data("Производитель создал товар: " + to\_string(product) + "\n\n");

}

else

{

print\_data("Поток производителя завершается. Всего было произведено: " + to\_string(number\_of\_product\_produced) + " товара(ов).\n\n");

// Разблокируем мьютекс

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

     return NULL;

}

print\_data("Производитель пытается поместить товар " + to\_string(product) + " в хранилище...\n\n");

         // Проверяем, есть ли свободные места на складе (while, а не if - защита от ложных пробуждений)

         while(warehouse.size() == NUM\_OF\_SLOTS)

         {

print\_data("Хранилище заполнено! Производитель ожидает появления свободного места.\n\n");

// Блокируем поток-производитель до момента, пока какой-либо потребитель не освободит место в хранилище

             status = pthread\_cond\_wait(&cond, &mutex);

             if(status != 0)

             {

                 err\_exit(status, "Cannot wait on condition variable");

             }

         }

// Допускаем создание товара только если производство не завершено

if(number\_of\_product\_produced < CAPACITY)

{

// Помещаем произведенный товар в хранилище и инкрементируем счетчик

warehouse.push(product);

number\_of\_product\_produced ++;

print\_data("Производитель успешно добавил товар " + to\_string(product) + " в хранилище.\nТоваров в хранилище: " + to\_string(warehouse.size()) + " из " + to\_string(NUM\_OF\_SLOTS) + ".\n\n");

// Посылаем сигнал, что на складе появился новый товар

status = pthread\_cond\_signal(&cond);

if(status != 0)

{

err\_exit(status, "Cannot send signal");

}

}

// Обеспечиваем своевременное завершение потока-производителя

if(number\_of\_product\_produced == CAPACITY)

{

print\_data("Поток производителя завершается. Всего было произведено: " + to\_string(number\_of\_product\_produced) + " товара(ов).\n\n");

// Разблокируем мьютекс

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

     return NULL;

}

// Разблокируем мьютекс

         status = pthread\_mutex\_unlock(&mutex);

         if(status != 0)

         {

             err\_exit(status, "Cannot unlock mutex");

         }

     }

}

// Функция, выполняемая потоком-потребителем

void \*consumer(void \*arg)

{

int consumer\_number = static\_cast<ThreadParams\*>(arg)->index;

int consumer\_time = static\_cast<ThreadParams\*>(arg)->time;

     int status;

// Будем сохранять все полученные потребителем товары

vector<int> products;

print\_data("Поток Потребителя №" + to\_string(consumer\_number) + " начинает свою работу!\n\n");

     while(true)

     {

// Задержка перед очередным потреблением

sleep(consumer\_time);

// Пытаемся получить доступ к общему ресурсу

         status = pthread\_mutex\_lock(&mutex);

         if(status != 0)

         {

             err\_exit(status, "Cannot lock mutex");

         }

print\_data("Потребитель №" + to\_string(consumer\_number) + " пытается получить товар из хранилища...\n\n");

         // Проверяем, есть ли товары на складе

         while(warehouse.empty())

         {

// Склад опустел и производство окончено

if(number\_of\_product\_produced == CAPACITY)

{

string products\_string = "[ ";

for(int product : products)

{

products\_string += to\_string(product) + " ";

}

products\_string += "]";

print\_data("Поток потребителя №" + to\_string(consumer\_number) + " завершается, так как производство завершено.\nСписок потребленных товаров: " + products\_string + "\n\n");

status = pthread\_cond\_signal(&cond);

if(status != 0)

{

err\_exit(status, "Cannot send signal");

}

status = pthread\_mutex\_unlock(&mutex);

if(status != 0)

{

err\_exit(status, "Cannot unlock mutex");

}

return NULL;

}

print\_data("Хранилище пустое! Потребитель №" + to\_string(consumer\_number) + " ожидает появление нового товара.\n\n");

// Ждем пополнение склада

             status = pthread\_cond\_wait(&cond, &mutex);

             if(status != 0)

             {

                 err\_exit(status, "Cannot wait on condition variable");

             }

         }

        // Берем товар со склада

        products.push\_back(warehouse.front());

         warehouse.pop();

print\_data("Потребитель №" + to\_string(consumer\_number) + " успешно получил товар " + to\_string(products.back()) + " из хранилища.\nТоваров в хранилище: " + to\_string(warehouse.size()) + " из " + to\_string(NUM\_OF\_SLOTS) + ".\n\n");

// Посылаем сигнал о потребленном товаре

         status = pthread\_cond\_signal(&cond);

         if(status != 0)

         {

             err\_exit(status, "Cannot send signal");

         }

// Разблокируем мьютекс

         status = pthread\_mutex\_unlock(&mutex);

         if(status != 0)

         {

             err\_exit(status, "Cannot unlock mutex");

         }

     }

}

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

{

srand(time(0));

int num\_of\_consumers = (argc > 1 ? stoi(argv[1]) :NUM\_OF\_CONSUMERS\_DEFAULT);

int consumer\_time = (argc > 2 ? stoi(argv[2]) : CONSUMER\_TIME\_DEFAULT);

int producer\_time = (argc > 3 ? stoi(argv[3]) : PRODUCER\_TIME\_DEFAULT);

// Храним потоки производителя и потребителей в одном массиве

vector<pthread\_t> threads(num\_of\_consumers + 1);

vector<ThreadParams> thread\_params(num\_of\_consumers + 1);

     int status;

// Инициализируем условную переменную

     status = pthread\_cond\_init(&cond, NULL);

     if(status != 0)

     {

         err\_exit(status, "Cannot initialize condition variable");

     }

// Инициализируем мьютекс

     status = pthread\_mutex\_init(&mutex, NULL);

     if(status != 0)

     {

         err\_exit(status, "Cannot initialize mutex");

     }

// Заполняем параметры потока-производителя и запускаем его

thread\_params[0] = { 0, producer\_time };

     status = pthread\_create(&threads[0], NULL, producer, &thread\_params[0]);

     if(status != 0)

     {

         err\_exit(status, "Cannot create a producer thread");

     }

// Аналогично делаем с потоками-потребителями

     for(int i = 1; i < threads.size(); i++)

{

thread\_params[i] = { i, consumer\_time };

status = pthread\_create(&threads[i], NULL, consumer, &thread\_params[i]);

if(status != 0)

{

err\_exit(status, "Cannot create consumer thread");

}

}

// Ожидаем завершения всех потоков

for(int i = 0; i < threads.size(); i++)

{

status = pthread\_join(threads[i], NULL);

if(status != 0)

{

err\_exit(status, "Cannot join a thread");

}

}

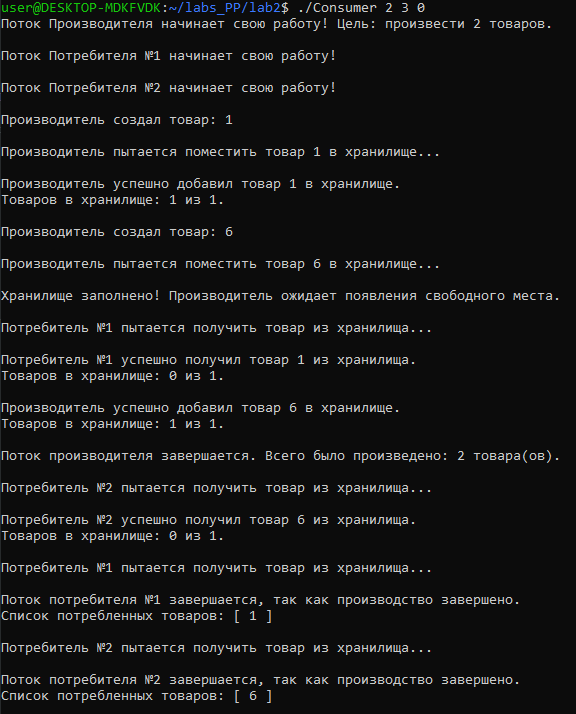
     pthread\_mutex\_destroy(&mutex);

     pthread\_cond\_destroy(&cond);

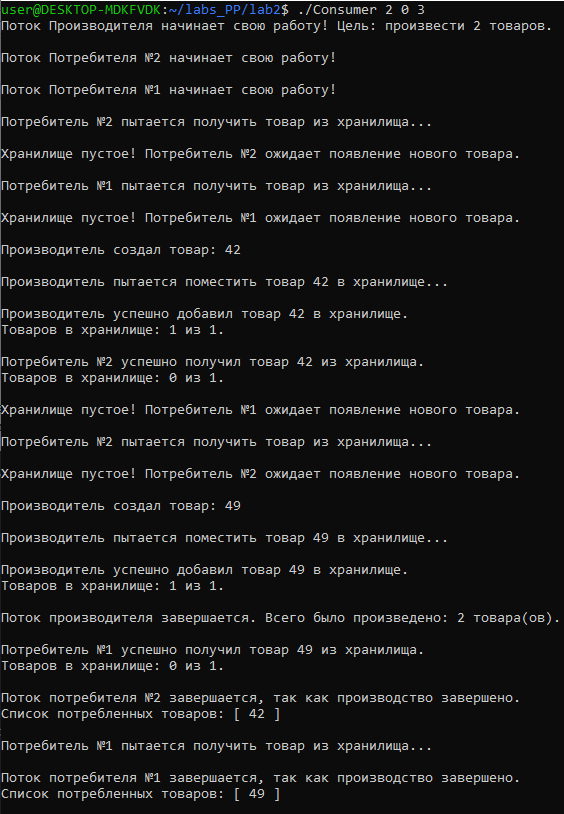
     return 0;

}

Выполним программу (емкость хранилища 1, общее число товаров, которые нужно произвести - 2). Сделаем так, чтобы производитель ожидал свободного места в хранилище:



А теперь пусть потребители ожидают пополнение пустого хранилища:



В процессе выполнения программы на экран выводится вся необходимая информация о действиях производителя и потребителей. В качестве дополнения на экран выводятся число товаров в хранилище после взаимодействия с ним одним из потоков, а также результирующий список потребленных каждым потребителем товаров.

Добавим измерение общего времени выполнения всех потоков в секундах. Будем менять число потоков-потребителей и целевое число товаров. Емкость хранилища сделаем равной 10 для всех тестов. Уберем все задержки и поточные выводы на экран. Результаты:

| Число потоков-потребителей | Целевое число товаров | | | | |
| --- | --- | --- | --- | --- | --- |
| 10 | 100 | 1000 | 10000 | 100000 |
| 2 | 0.0010026 | 0.0075550 | 0.0768352 | 0.7508874 | 7.3735154 |
| 4 | 0.0012092 | 0.0081685 | 0.0709859 | 0.6873377 | 6.8668265 |
| 8 | 0.0016037 | 0.0086806 | 0.0717514 | 0.6923128 | 6.8268061 |
| 16 | 0.0020071 | 0.0091786 | 0.0714254 | 0.6916352 | 6.8898877 |

На основе измерений можно сделать следующие выводы:

* 1. Время выполнения увеличивается с увеличением целевого числа товаров. Это ожидаемо.
  2. Увеличение числа потоков-потребителей не всегда приводит к уменьшению времени выполнения. Время выполнения немного увеличивается при увеличении числа потоков-потребителей для целевого числа товаров 10 и 100. Однако, для большего числа товаров (1000, 10000 и 100000), время выполнения сначала уменьшается, а затем остается примерно одинаковым при дальнейшем увеличении числа потоков-потребителей.

Существует оптимальное число потоков-потребителей, которое почти при любом числе товаров минимизирует время выполнения. Это число равно 4.

Как было замечено ранее, при увеличении числа потребителей общее время выполнения потоков в большинстве случаев не уменьшается. Это вызвано тем, что в процессе измерений была убрана любая задержка (время обработки товара), из-за чего большую часть времени занимает координация между потоками, а не сам процесс потребления.

Для доказательства этого предположения приведем замеры времени работы потоков, добавив секунду на потребление товара каждым потребителем (производитель продолжает производить товары с нулевой задержкой). Целевое число товаров: 10.

| Число потоков-потребителей | Время выполнения всех потоков |
| --- | --- |
|
| 2 | 6.0017296 |
| 4 | 4.0013077 |
| 8 | 3.0008338 |
| 16 | 2.0008017 |

Это подтверждает, что увеличение количества потоков-потребителей может значительно уменьшить общее время выполнения, особенно когда время обработки товара (полезная работа) составляет значительную часть общего времени выполнения.