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

I. Цель работы

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

II. 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:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$ ./Task_2
Task number: 0
Task number: 1
Task number: 3
Task number: 4
Task number: 5
Task number: 6
Task number: 7
Task number: 8
Task number: 9
Task number: 2
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$
```

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:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$ ./Task_3
Tasks done by thread 1: 0 2 4 6 8
Tasks done by thread 2: 1 3 5 7 9
```

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:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$ ./Task_3
Tasks done by thread 1: 0 2 4 6 8
Tasks done by thread 2: 0 2 4 6 8
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$ ./Task_3
Tasks done by thread 1: 0 2 4 7 9
Tasks done by thread 2: 0 2 4 6 8 9
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$
```

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:

```
Mutex test time:     0.0000003330
Spinlock test time:  0.0000001290
```

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:

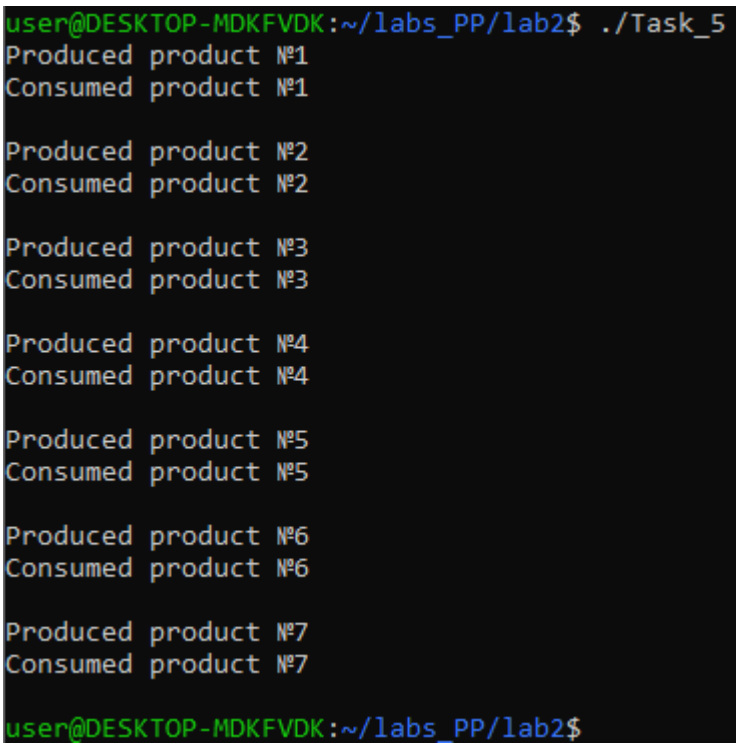
```
#include <iostream>
#include <pthread.h>
#include <cstdlib>
#include <cstring>
#include <unistd.h>

#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):

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$ ./Task_6 3 3 0
Enter 3 lines:
-----------------------------------------------------
Hello world!
AEoiea
I am Fine
-----------------------------------------------------

MapReduce execution time: 0.000494897

The number of vowels in a set of strings:
a = 3
e = 4
i = 3
o = 3
u = 0
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$
```

Пример с автозаполнением:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$ ./Task_6 3 3 1
The set contains 3 lines of the form: Hello World! How do you like this program?

MapReduce execution time: 0.00031938

The number of vowels in a set of strings:
a = 3
e = 6
i = 6
o = 18
u = 3
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$
```

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:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$ ./Consumer 2 3 0
Поток Производителя начинает свою работу! Цель: произвести 2 товаров.

Поток Потребителя №1 начинает свою работу!

Поток Потребителя №2 начинает свою работу!

Производитель создал товар: 1

Производитель пытается поместить товар 1 в хранилище...

Производитель успешно добавил товар 1 в хранилище.
Товаров в хранилище: 1 из 1.

Производитель создал товар: 6

Производитель пытается поместить товар 6 в хранилище...

Хранилище заполнено! Производитель ожидает появления свободного места.

Потребитель №1 пытается получить товар из хранилища...

Потребитель №1 успешно получил товар 1 из хранилища.
Товаров в хранилище: 0 из 1.

Производитель успешно добавил товар 6 в хранилище.
Товаров в хранилище: 1 из 1.

Поток производителя завершается. Всего было произведено: 2 товара(ов).

Потребитель №2 пытается получить товар из хранилища...

Потребитель №2 успешно получил товар 6 из хранилища.
Товаров в хранилище: 0 из 1.

Потребитель №1 пытается получить товар из хранилища...

Поток потребителя №1 завершается, так как производство завершено.
Список потребленных товаров: [ 1 ]

Потребитель №2 пытается получить товар из хранилища...

Поток потребителя №2 завершается, так как производство завершено.
Список потребленных товаров: [ 6 ]
```

А теперь пусть потребители ожидают пополнение пустого хранилища:

```
user@DESKTOP-MDKFVDK:~/labs_PP/lab2$ ./Consumer 2 0 3
Поток Производителя начинает свою работу! Цель: произвести 2 товаров.

Поток Потребителя №2 начинает свою работу!

Поток Потребителя №1 начинает свою работу!

Потребитель №2 пытается получить товар из хранилища...

Хранилище пустое! Потребитель №2 ожидает появление нового товара.

Потребитель №1 пытается получить товар из хранилища...

Хранилище пустое! Потребитель №1 ожидает появление нового товара.

Производитель создал товар: 42

Производитель пытается поместить товар 42 в хранилище...

Производитель успешно добавил товар 42 в хранилище.
Товаров в хранилище: 1 из 1.

Потребитель №2 успешно получил товар 42 из хранилища.
Товаров в хранилище: 0 из 1.

Хранилище пустое! Потребитель №1 ожидает появление нового товара.

Потребитель №2 пытается получить товар из хранилища...

Хранилище пустое! Потребитель №2 ожидает появление нового товара.

Производитель создал товар: 49

Производитель пытается поместить товар 49 в хранилище...

Производитель успешно добавил товар 49 в хранилище.
Товаров в хранилище: 1 из 1.

Поток производителя завершается. Всего было произведено: 2 товара(ов).

Потребитель №1 успешно получил товар 49 из хранилища.
Товаров в хранилище: 0 из 1.

Поток потребителя №2 завершается, так как производство завершено.
Список потребленных товаров: [ 42 ]

Потребитель №1 пытается получить товар из хранилища...

Поток потребителя №1 завершается, так как производство завершено.
Список потребленных товаров: [ 49 ]
```

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:

	Целевое число товаров

Число потоков-потребителей	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.