Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3 по курсу**

**«Операционные системы»**

Студент: Орусский Вячеслав Русланович

Группа: М8О-206Б-21

Вариант: 18

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2023

**Содержание**

**[Репозиторий](#_Toc133018213)** [3](#_Toc133018213)

[**Постановка задачи** 3](#_Toc133018214)

[**Цель работы** 3](#_Toc133018215)

[**Задание** 3](#_Toc133018216)

[**Общие сведения о программе** 3](#_Toc133018217)

[**Общий метод и алгоритм решения** 4](#_Toc133018218)

[**Исходный код** 4](#_Toc133018219)

[**Демонстрация работы программы** 6](#_Toc133018220)

[**Выводы** 6](#_Toc133018221)

**Репозиторий**

[Ссылка](https://github.com/RoKivals/OperatingSystems/tree/master/Lab%233/src)

**Постановка задачи**

**Цель работы**

* Управление потоками в ОС
* Обеспечение синхронизации между потоками

**Задание**

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработке использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение потоков должно быть задано ключом запуска вашей программы.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

В отчете привести исследование зависимости ускорения и эффективности алгоритма от входящих данных и количества потоков. Получившиеся результаты необходимо объяснить. В моём варианте необходимо найти образец в строке наивным алгоритмом.

**Общие сведения о программе**

Программа компилируется из файла main.cpp. В программе используются следующие системные вызовы:

1. pthread\_mutex\_init(…) – инициализировать мьютекс
2. pthread\_mutex\_lock(…) – «закрыть» мьютекс
3. pthread\_mutex\_unlock(…) – «открыть» мьютекс
4. pthread\_create(…) – создать поток
5. pthread\_join(…) - ожидание завершения потока, заданного параметром

**Общий метод и алгоритм решения**

Для начала считываем количество потоков, переданное на входной поток программе. Далее подсчитываем реальное количество потоков, которое мы будем использовать по следующему алгоритму: если введённое количество потоков меньше или равно размеру строки минус размер образца плюс 1 (это крайний случай, когда каждый поток будет обрабатывать вхождение образца только один раз, иначе остальным потокам нечего будет делать), то мы принимаем введенное количество потоков и вычисляем размер области поиска для каждого из потоков как размер строки делённый на количество потоков. Иначе мы делаем такое количество потоков, какое и описано в крайнем случае выше (размер строки минус размер образца + 1). Далее создаём это количество потоков, в каждом из которых производим поиск образца в строке наивным поиском (посимвольно сравнивая текущий фрагмент строки с образцом, смещаясь на 1 символ при каждой итерации).

**Исходный код**

#include <bits/stdc++.h>  
  
pthread\_mutex\_t mutex; *// Переменная для Mutex  
  
// структура аргументов потока  
struct* threadArgs {  
 std::string mainString;  
 std::string pattern;  
 *int* index;  
 size\_t size;  
};  
  
*// Функция наивного поиска подстроки в строке  
void* NativeSearch(*void* \*inArgs) {  
 pthread\_mutex\_lock(&mutex);  
 threadArgs args = \*(threadArgs \*) inArgs; *// приводим принятые аргументы к типу структуры аргументов* std::string mainString = args.mainString;  
 std::string pattern = args.pattern;  
 *int* index = args.index; *// Номер текущего потока* size\_t size = args.size;  
 size\_t pattern\_len = pattern.size(); *// Длина подстроки  
  
 // index \* size; index \* size + size - начало и конец поиска для потока  
 for* (*int* i(index \* (*int*) size); i < (index \* (*int*) size + (*int*) size); ++i) {  
 *int* j(0);  
 *for* (; j < pattern\_len; ++j) {  
 *if* (mainString[i + j] != pattern[j]) {  
 *break*; *// если какой-то символ подстроки и строки не совпал, то переходим к следующему i* }  
 }  
  
 *// Если подстрока полностью совпала, то выводим индекс вхождения  
 if* (j == pattern\_len) {  
 std::cout << "Thread [" << index << "] " << "Pattern found at position: " << i << std::endl;  
 }  
 }  
 free(inArgs);  
 pthread\_mutex\_unlock(&mutex); *// открываем мьютекс для других потоков*}  
  
*int* main(*int* argc, *char* \*\*argv) {  
 size\_t inThreads = 1; *// переменная для введенного количества потоков  
  
 // Кол-во потоков не указано, используем по умолчанию 1  
 if* (argc != 2) {  
 std::cerr << "Incorrect num of arguments, expected thread count" << std::endl;  
 std::cerr << "Setting thread count to 1" << std::endl;  
 } *else* {  
 inThreads = std::stoi(argv[1]); *// преобразуем строку в число* }  
  
 *// Основная строка* std::string mainString;  
 *// Подстрока* std::string pattern;  
 std::cout << "Input main string: ";  
 std::cin >> mainString;  
 std::cout << "Input a pattern string to search in main string: ";  
 std::cin >> pattern;  
  
 *// Кол-во потоков и размер области поиска для каждого* size\_t threadCount = inThreads, size;  
 *// Если кол-во потоков меньше кол-ва итераций по строке (n - m)  
 if* (inThreads <= mainString.size() - pattern.size() + 1) {  
 size = mainString.size() / inThreads;  
 } *else* {  
 *// Каждый поток обрабатывает ровно одну итерацию по строке* threadCount = mainString.size() - pattern.size() + 1;  
 *// 1 поток - 1 вхождение* size = 1;  
 }  
  
 pthread\_t threads[threadCount];  
 *int* i(0);  
 pthread\_mutex\_init(&mutex, *nullptr*);  
 *for* (; i < threadCount; i++) {  
 pthread\_mutex\_lock(&mutex); *// закрываем мьютекс  
 auto* \*arguments = *new* threadArgs;  
 arguments->mainString = mainString; *// заполняем структуру данными* arguments->pattern = pattern;  
 arguments->index = i;  
 arguments->size = size;  
 *// запускаем поток, кастуем функцию к общему указателю, который возвращает и принимает общий указатель (pretty!)  
 if* (pthread\_create(&threads[i], *nullptr*, *reinterpret\_cast*<*void* \*(\*)(*void* \*)>(&NativeSearch), arguments) != 0) {  
 std::cerr << "Creating thread error" << std::endl;  
 }  
 pthread\_mutex\_unlock(&mutex);  
 }  
  
 *for* (i = 0; i < threadCount; i++) {  
 *if* (pthread\_join(threads[i], *nullptr*) != 0) {  
 *// тут мы ждём, когда потоки завершатся. Если кто-то вернул код != 0 - выводим ошибку* std::cerr << "Thread " << i << " work confused" << std::endl;  
 }  
 }  
 pthread\_mutex\_destroy(&mutex);  
 *return* 0;  
}

**Демонстрация работы программы**

Строка, в которой ищем: "catdogcathousecatdoga"  
Образец, который ищем: "cat"

slava@DESKTOP-9JJF73M MINGW64 ./lab3 4

[2] Pattern found at position: 14

[1] Pattern found at position: 6

[0] Pattern found at position: 0

slava@DESKTOP-9JJF73M MINGW64 ./lab3 1

[0] Pattern found at position: 0

[0] Pattern found at position: 6

[0] Pattern found at position: 14

slava@DESKTOP-9JJF73M MINGW64 ./lab3 1000

[0] Pattern found at position: 0

[6] Pattern found at position: 6

[14] Pattern found at position: 14

**Выводы**

Проделав данную работу, я научилась управлять потоками в ОС, обеспечивать синхронизацию потоков. Сделала вывод, что синхронизация потоков очень важна, так как при работе с «несинхронизированными» потоками мы можем получить неожидаемое поведение. Так, например, до использования мьютексов в моей лабораторной работе, несколько потоков могли начать одновременно выводить данные и вывод данных смешивался, превращаясь в кашу. Так же могли быть проблемы с доступом к памяти (все потоки обращались бы к одним и тем же адресам), но эту проблему удалось решить ещё до использования мьютексов путём выделения памяти для каждого из процессов.