1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. **Институт кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА № 3**

1. «Механизмы многопоточности»
2. по дисциплине «Операционные системы»
3. Выполнил
4. студент гр. 4851003/10002 Тоцкий В.

<*подпись*>

* + 1. Преподаватель Крундышев В.М.

<*подпись*>

Санкт-Петербург

1. 2023

**ЦЕЛЬ РАБОТЫ**

Изучить принципы разработки многопоточных программ, изучить программный интерфейс операционных систем для организации многопоточности, получить навыки организации взаимодействия потоков в многопоточных программах.

**ХОД РАБОТЫ**

В данной лабораторной работе главной задачей являлось изучение механизмов многопоточности, для это были написаны 4 программы, которые на примере решения популярных задач и сортировок было показано, как работают механизмы многопоточности.

1. **Вычисления (expr.cpp).**

Для начала рассмотрим блок-схему работы данного алгоритма (см. Рисунок 1)

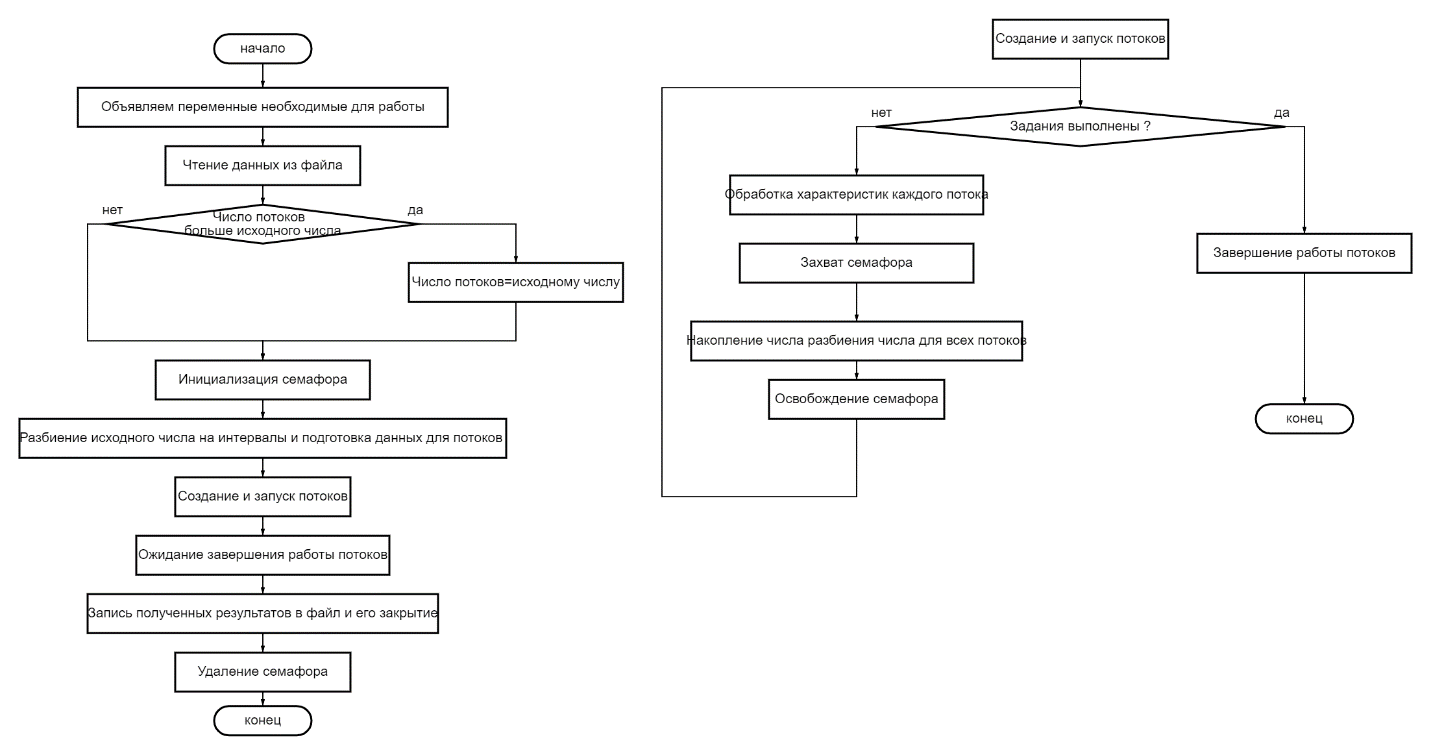


Рисунок 1 – Блок-схема работы алгоритма expr.cpp.

|  |  |  |
| --- | --- | --- |
| Название функции | Аргументы | Описание |
| void readFile | int \*threadCount, int \*number | // функция для чтения данных из файла |
| int countSplits | int id | // функция для вычисления количества разбиений, обрабатываемых одним потоком |
| void \*threadFunction | void \*param | // функция, которая будет выполняться в каждом потоке |
| void divideData | int interval, int threadCount, int number) | // функция для разделения данных между потоками |

В таблице 1 представлены функции, которые использовались в expr.cpp.

Таблица 1 - Функции, реализованные в программе expr.cpp.

1. **Быстрая сортировка (qsort.cpp).**

На рисунке 2 представлена блок-схема алгоритма qsort.cpp.

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 2 – Блок-схема работы алгоритма qsort.cpp.

В таблице 2 представлены функции, которые использовались в qsort.cpp.

Таблица 2 - Функции, реализованные в программе qsort.cpp.

|  |  |  |
| --- | --- | --- |
| Название функции | Аргументы | Описание |
| int partition | vector<int>& arr, int low, int high | // Функция разделения для алгоритма быстрой сортировки |
| void quicksort | vector<int>& arr, int low, int high | // Функция быстрой сортировки |
| void thread\_handler | vector<int>& arr | // Функция обработки потока |

1. **Сортировка слияниями (MergeSort, msort,cpp)**

На рисунке 3 представлена блок-схема алгоритма msort.cpp.

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 3 – Блок-схема алгоритма msort.cpp.

1. **Философы (phil.cpp)**

На рисунке 4 представлена блок-схема алгоритма phil.cpp.

Изображение выглядит как диаграмма

Автоматически созданное описание

Рисунок 4 – Блок-схема алгоритма phil.cpp

В таблице 4 представлены функции, которые использовались в phil.cpp.

Таблица 4 - Функции, реализованные в программе phil.cpp.

|  |  |  |
| --- | --- | --- |
| Название функции | Аргументы | Описание |
| void init |  | // Функция инициализации |
| static unsigned long long ticks\_to\_ms() |  | // Функция конвертации системных тиков в миллисекунды |
| DWORD WINAPI thread | LPVOID p | // Функция потока |

**РЕЗУЛЬТАТЫ РАБОТЫ**

После проведения тестирования программ были составлены гистограммы сравнения времени работы для разного числа потоков. Тестирование программ на время проводилось на процессоре AMD Ryzen 9 5900 HX.

1. Вычисления (expr.cpp).

Таблица 5 – Примеры работы программы expr.cpp

|  |  |
| --- | --- |
| Ввод | Вывод |
| 1 35 | 1 35 14882 |
| 3 40 | 3 40 37337 |
| 8 50 | 8 50 244225 |

Рисунок 5 – Гистограмма тестирования expr.cpp

Быстрая сортировка (qsort.cpp).

Таблица 6 – Примеры работы программы qsort.cpp

|  |  |
| --- | --- |
| Ввод | Вывод |
| 4  6  8 9 7 1 2 4 | 4  6  1 2 4 7 8 9 |

Рисунок 6 – Гистограмма тестирования qsort.cpp

Сортировка слияниями (msort,cpp)

Рисунок 7 – Гистограмма тестирования msort.cpp

1. Философы (phil.cpp)

Результаты тестирования философов на 2000 20:

31:1:T->E

31:3:T->E

47:1:E->T

47:3:E->T

47:2:T->E

47:4:T->E

62:2:E->T

62:4:E->T

62:3:T->E

62:5:T->E

94:3:E->T

94:5:E->T

94:4:T->E

94:1:T->E

109:4:E->T

109:1:E->T

109:5:T->E

109:2:T->E

140:5:E->T

140:2:E->T

140:1:T->E

140:3:T->E

156:1:E->T

156:3:E->T

156:2:T->E

156:4:T->E

172:2:E->T

172:4:E->T

172:3:T->E

172:5:T->E

203:3:E->T

203:5:E->T

203:4:T->E

203:1:T->E

219:4:E->T

219:1:E->T

219:5:T->E

219:2:T->E

250:5:E->T

250:2:E->T

250:1:T->E

250:3:T->E

265:1:E->T

265:3:E->T

265:2:T->E

265:4:T->E

281:2:E->T

281:4:E->T

281:3:T->E

281:5:T->E

312:3:E->T

312:5:E->T

312:4:T->E

312:1:T->E

328:4:E->T

328:1:E->T

328:5:T->E

328:2:T->E

359:5:E->T

359:2:E->T

359:1:T->E

359:3:T->E

375:1:E->T

375:3:E->T

375:2:T->E

375:4:T->E

406:2:E->T

406:4:E->T

406:3:T->E

406:5:T->E

422:3:E->T

422:5:E->T

422:4:T->E

422:1:T->E

453:4:E->T

453:1:E->T

453:5:T->E

453:2:T->E

469:5:E->T

469:2:E->T

469:1:T->E

469:3:T->E

484:1:E->T

484:3:E->T

484:2:T->E

484:4:T->E

515:2:E->T

515:4:E->T

515:3:T->E

515:5:T->E

547:3:E->T

547:5:E->T

547:4:T->E

547:1:T->E

562:4:E->T

562:1:E->T

562:5:T->E

562:2:T->E

594:5:E->T

594:2:E->T

594:1:T->E

594:3:T->E

609:1:E->T

609:3:E->T

609:2:T->E

609:4:T->E

640:2:E->T

640:4:E->T

640:3:T->E

640:5:T->E

656:3:E->T

656:5:E->T

656:4:T->E

656:1:T->E

687:4:E->T

687:1:E->T

687:5:T->E

687:2:T->E

703:5:E->T

703:2:E->T

703:1:T->E

703:3:T->E

719:1:E->T

719:3:E->T

719:2:T->E

719:4:T->E

750:2:E->T

750:4:E->T

750:3:T->E

750:5:T->E

765:3:E->T

765:5:E->T

765:4:T->E

765:1:T->E

797:4:E->T

797:1:E->T

797:5:T->E

797:2:T->E

812:5:E->T

812:2:E->T

812:1:T->E

812:3:T->E

844:1:E->T

844:3:E->T

844:2:T->E

844:4:T->E

859:2:E->T

859:4:E->T

859:3:T->E

859:5:T->E

890:3:E->T

890:5:E->T

890:4:T->E

890:1:T->E

906:4:E->T

906:1:E->T

906:5:T->E

906:2:T->E

937:5:E->T

937:2:E->T

937:1:T->E

937:3:T->E

953:1:E->T

953:3:E->T

953:2:T->E

953:4:T->E

984:2:E->T

984:4:E->T

984:3:T->E

984:5:T->E

1000:3:E->T

1000:5:E->T

1000:4:T->E

1000:1:T->E

1031:4:E->T

1031:1:E->T

1031:5:T->E

1031:2:T->E

1047:5:E->T

1047:2:E->T

1047:1:T->E

1047:3:T->E

1078:1:E->T

1078:3:E->T

1078:2:T->E

1078:4:T->E

1094:2:E->T

1094:4:E->T

1094:3:T->E

1094:5:T->E

1109:3:E->T

1109:5:E->T

1125:4:T->E

1125:1:T->E

1140:4:E->T

1140:1:E->T

1140:5:T->E

1140:2:T->E

1156:5:E->T

1156:2:E->T

1156:1:T->E

1156:3:T->E

1187:1:E->T

1187:3:E->T

1187:2:T->E

1187:4:T->E

1203:2:E->T

1203:4:E->T

1203:3:T->E

1203:5:T->E

1234:3:E->T

1234:5:E->T

1234:4:T->E

1234:1:T->E

1250:4:E->T

1250:1:E->T

1250:5:T->E

1250:2:T->E

1281:5:E->T

1281:2:E->T

1281:1:T->E

1281:3:T->E

1297:1:E->T

1297:3:E->T

1297:2:T->E

1297:4:T->E

1328:2:E->T

1328:4:E->T

1328:3:T->E

1328:5:T->E

1344:3:E->T

1344:5:E->T

1344:4:T->E

1344:1:T->E

1359:4:E->T

1359:1:E->T

1359:5:T->E

1359:2:T->E

1390:5:E->T

1390:2:E->T

1390:1:T->E

1390:3:T->E

1406:1:E->T

1406:3:E->T

1406:2:T->E

1406:4:T->E

1437:2:E->T

1437:4:E->T

1437:3:T->E

1437:5:T->E

1453:3:E->T

1453:5:E->T

1453:4:T->E

1453:1:T->E

1484:4:E->T

1484:1:E->T

1484:5:T->E

1484:2:T->E

1500:5:E->T

1500:2:E->T

1500:1:T->E

1500:3:T->E

1531:1:E->T

1531:3:E->T

1531:2:T->E

1531:4:T->E

1547:2:E->T

1547:4:E->T

1547:3:T->E

1547:5:T->E

1562:3:E->T

1562:5:E->T

1562:4:T->E

1562:1:T->E

1594:4:E->T

1594:1:E->T

1594:5:T->E

1594:2:T->E

1609:5:E->T

1609:2:E->T

1609:1:T->E

1609:3:T->E

1640:1:E->T

1640:3:E->T

1640:2:T->E

1640:4:T->E

1656:2:E->T

1656:4:E->T

1656:3:T->E

1656:5:T->E

1687:3:E->T

1687:5:E->T

1687:4:T->E

1687:1:T->E

1703:4:E->T

1703:1:E->T

1703:5:T->E

1703:2:T->E

1719:5:E->T

1719:2:E->T

1734:1:T->E

1734:3:T->E

1750:1:E->T

1750:3:E->T

1750:2:T->E

1750:4:T->E

1765:2:E->T

1765:4:E->T

1765:3:T->E

1765:5:T->E

1797:3:E->T

1797:5:E->T

1797:4:T->E

1797:1:T->E

1812:4:E->T

1812:1:E->T

1812:5:T->E

1812:2:T->E

1844:5:E->T

1844:2:E->T

1844:1:T->E

1844:3:T->E

1859:1:E->T

1859:3:E->T

1859:2:T->E

1859:4:T->E

1890:2:E->T

1890:4:E->T

1890:3:T->E

1890:5:T->E

1906:3:E->T

1906:5:E->T

1906:4:T->E

1906:1:T->E

1922:4:E->T

1922:1:E->T

1922:5:T->E

1922:2:T->E

1953:5:E->T

1953:2:E->T

1953:1:T->E

1953:3:T->E

1969:1:E->T

1969:3:E->T

1969:2:T->E

1969:4:T->E

2000:2:E->T

2000:4:E->T

Таблица распределения процессорного времени между философами.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Испытание** | **Время теста** | **Время поедания** | **Время раздумывания** | **Количество квантов поедания** | **Количество квантов раздумывания** |
| 1 | 200мс | 50мс | 50мс | 4 | 4 |
| 2 | 200мс | 100мс | 50мс | 2 | 4 |
| 3 | 200мс | 100мс | 100мс | 2 | 2 |
| 4 | 400мс | 100мс | 100мс | 4 | 4 |
| 5 | 400мс | 200мс | 100мс | 2 | 4 |
| 6 | 400мс | 200мс | 200мс | 2 | 2 |
| 7 | 1000мс | 250мс | 250мс | 4 | 4 |
| 8 | 1000мс | 500мс | 250мс | 2 | 4 |
| 9 | 1000мс | 500мс | 500мс | 2 | 2 |
| 10 | 2000мс | 500мс | 500мс | 4 | 4 |
| 11 | 2000мс | 1000мс | 500мс | 2 | 4 |
| 12 | 2000мс | 1000мс | 1000мс | 2 | 2 |
| 13 | 4000мс | 1000мс | 1000мс | 4 | 4 |
| 14 | 4000мс | 2000мс | 1000мс | 2 | 4 |
| 15 | 4000мс | 2000мс | 2000мс | 2 | 2 |
| 16 | 20000мс | 5000мс | 5000мс | 4 | 4 |
| 17 | 20000мс | 10000мс | 5000мс | 2 | 4 |
| 18 | 20000мс | 10000мс | 10000мс | 2 | 2 |

Исходя из таблицы, можно сделать вывод, что увеличение общей продолжительности теста и времени поедания/раздумывания приводит к увеличению количества "квантов" выполнения действий у каждого философа. Однако, распределение процессорного времени между философами не всегда было равномерным, особенно в случаях, когда время поедания/раздумывания было неравномерным. Таким образом, для обеспечения равномерности распределения процессорного времени между философами необходимо подбирать соответствующие параметры времени поедания/раздумывания, а также общей продолжительности теста.

**ВЫВОДЫ**

В ходе выполнения данной лабораторной были изучены принципы организации многопоточности, используя различные способы синхронизации. такие как Mutex, семафоры, а также критическая секция, в ОС Windows и Linux. Исходя из полученных гистограмм, можно сказать, что чем больше использовалось потоков, тем меньше по времени выполнялась программа.

**ПРИЛОЖЕНИЕ**

1. *Листинг кода expr.cpp*

#include <iostream> // подключение библиотеки ввода/вывода

#include <fstream> // подключение библиотеки работы с файлами

#include <cstdio> // подключение библиотеки работы со стандартным вводом/выводом

#include <list> // подключение библиотеки работы со списками

#include <pthread.h> // подключение библиотеки работы с потоками

#include <malloc.h> // подключение библиотеки работы с памятью

#include <time.h> // подключение библиотеки работы со временем

#include <semaphore.h> // подключение библиотеки работы с семафорами

using namespace std; // использование пространства имен std

sem\_t semaphore; // объявление переменной семафора

volatile int totalCount; // объявление переменной для хранения общего количества разбиений

struct ThreadInfo // определение структуры для передачи информации о потоке

{

volatile int limit; // граница разбиений, обрабатываемых потоком

volatile int \*parts; // указатель на массив слагаемых

volatile int length; // длина массива слагаемых

};

volatile ThreadInfo threadInfo[100]; // объявление массива структур ThreadInfo

void readFile(int \*threadCount, int \*number) // функция для чтения данных из файла

{

FILE \*inputFile = fopen("input.txt", "r"); // открытие файла для чтения

if (inputFile == NULL) // если не удалось открыть файл, то выводится сообщение об ошибке и функция завершает работу

{

printf("FILE OPEN ERROR");

return;

}

fscanf(inputFile, "%d", threadCount); // чтение количества потоков из файла

fscanf(inputFile, "%d", number); // чтение числа, которое нужно разбить на слагаемые

fclose(inputFile); // закрытие файла

}

int countSplits(int id) // функция для вычисления количества разбиений, обрабатываемых одним потоком

{

int end = threadInfo[id].limit; // граница разбиений, обрабатываемых потоком

int count = 0; // счетчик количества разбиений

int temp = 0; // временная переменная для хранения суммы слагаемых

int i = threadInfo[id].length; // индекс текущего слагаемого

if (threadInfo[id].parts[i] == 0) // если последнее слагаемое равно нулю, то не учитывается

{

i--;

}

int stop = threadInfo[id].limit; // граница разбиений, обрабатываемых потоком

while (1) // бесконечный цикл

{

count++; // увеличение счетчика количества разбиений

temp = 0; // сброс временной переменной

while (i > -1 && threadInfo[id].parts[i] == 1) // пока не достигнут начало массива и текущее слагаемое равно единице

{

temp += threadInfo[id].parts[i]; // добавляем текущее слагаемое к временной переменной

i--; // переходим к следующему слагаемому

}

if (i < 0) // если достигнуто начало массива, то возвращаем количество разбиений

{

return count;

}

threadInfo[id].parts[i] -= 1; // уменьшаем текущее слагаемое на единицу

temp++; // увеличиваем временную переменную на единицу

if (threadInfo[id].parts[0] == stop) // если достигнута граница разбиений, обрабатываемых потоком, то возвращаем количество разбиений

{

return count;

}

while (temp > threadInfo[id].parts[i]) // пока временная переменная больше текущего слагаемого

{

threadInfo[id].parts[i + 1] = threadInfo[id].parts[i]; // перемещаем текущее слагаемое в следующую ячейку

temp = temp - threadInfo[id].parts[i]; // вычитаем текущее слагаемое из временной переменной

i++; // переходим к следующему слагаемому

}

i++; // переходим к следующему слагаемому

threadInfo[id].parts[i] = temp; // устанавливаем текущее слагаемое в значение временной переменной

}

}

void \*threadFunction(void \*param) // функция, которая будет выполняться в каждом потоке

{

int id = (char \*)param - (char \*)0; // определение идентификатора потока по его адресу

int temp = countSplits(id); // вычисление количества разбиений, обрабатываемых потоком

sem\_wait(&semaphore); // захват семафора

totalCount += temp; // увеличение счетчика общего количества разбиений

sem\_post(&semaphore); // освобождение семафора

}

void divideData(int interval, int threadCount, int number) // функция для разделения данных между потоками

{

int remainder = number % threadCount;

int i = 0; // инициализация индекса текущего потока

int previous = 0; // предыдущее слагаемое

int current = 0; // текущее слагаемое

int first = interval; // первое слагаемое для первого потока

int second = number - interval; // второе слагаемое для первого потока

threadInfo[i].parts = (int \*)malloc(number \* sizeof(int) + 1); // выделение памяти для массива слагаемых первого потока

threadInfo[i].parts[0] = first; // установка первого элемента массива равным первому слагаемому

threadInfo[i].limit = 0; // граница разбиений, обрабатываемых первым потоком

previous = first; // предыдущее слагаемое устанавливаем равным первому

current = second; // текущее слагаемое устанавливаем равным второму

int j = 1; // индекс текущего элемента в массиве слагаемых

while (current > 0) // пока текущее слагаемое больше нуля

{

threadInfo[i].parts[j] = previous; // добавляем предыдущее слагаемое в массив

j++; // увеличиваем индекс текущего элемента в массиве

if (current < 2 \* previous) // если текущее слагаемое меньше удвоенного предыдущего, то устанавливаем предыдущее слагаемое равным разнице между текущим и предыдущим слагаемыми

{

previous = current - previous;

}

current = current - first; // уменьшаем текущее слагаемое на значение первого слагаемого

}

threadInfo[i].length = j - 1; // устанавливаем длину массива слагаемых первого потока

i++; // переходим к следующему потоку

first = (i + 1) \* interval; // устанавливаем первое слагаемое для следующего потока

second = number - (i + 1) \* interval; // устанавливаем второе слагаемое для следующего потока

while (second > first) // пока второе слагаемое больше первого

{

threadInfo[i].parts = (int \*)malloc(number \* sizeof(int) + 1); // выделение памяти для массива слагаемых текущего потока

threadInfo[i].parts[0] = first; // установка первого элемента массива равным первому слагаемому

threadInfo[i].limit = threadInfo[i - 1].parts[0]; // граница разбиений, обрабатываемых текущим потоком

previous = first; // предыдущее слагаемое устанавливаем рав

current = second; // текущее слагаемое устанавливаем равным второму

j = 1; // сбрасываем индекс текущего элемента в массиве

while (current > 0) // пока текущее слагаемое больше нуля

{

threadInfo[i].parts[j] = previous; // добавляем предыдущее слагаемое в массив

j++; // увеличиваем индекс текущего элемента в массиве

if (current < 2 \* previous) // если текущее слагаемое меньше удвоенного предыдущего, то устанавливаем предыдущее слагаемое равным разнице между текущим и предыдущим слагаемыми

{

previous = current - previous;

}

current = current - first; // уменьшаем текущее слагаемое на значение первого слагаемого

}

threadInfo[i].length = j - 1; // устанавливаем длину массива слагаемых текущего потока

i++; // переходим к следующему потоку

first = (i + 1) \* interval; // устанавливаем первое слагаемое для следующего потока

second = number - first; // устанавливаем второе слагаемое для следующего потока

}

while (i < threadCount - 1) // пока не достигнут последний поток

{

threadInfo[i].parts = (int \*)malloc(number \* sizeof(int) + 1); // выделение памяти для массива слагаемых текущего потока

threadInfo[i].parts[0] = first; // установка первого элемента массива равным первому слагаемому

threadInfo[i].parts[1] = second; // установка второго элемента массива равным второму слагаемому

if (i != 0) // если текущий поток не является первым, то устанавливаем границу разбиений, обрабатываемых текущим потоком

{

threadInfo[i].limit = threadInfo[i - 1].parts[0];

}

else // иначе устанавливаем границу равной нулю

{

threadInfo[i].limit = 0;

}

if (i + 1 == threadCount) // если текущий поток является последним, то уменьшаем границу на остаток от деления числа на количество потоков

{

threadInfo[i].limit -= remainder;

}

threadInfo[i].length = 1; // устанавливаем длину массива равной 1

i++; // переходим к следующему потоку

first = (i + 1) \* interval; // устанавливаем первое слагаемое для следующего потока

second = number - first; // устанавлив

}

threadInfo[i].parts = (int \*)malloc(number \* sizeof(int) + 1); // выделение памяти для массива слагаемых последнего потока

threadInfo[i].parts[0] = number; // установка первого элемента массива равным числу, которое требуется разбить

threadInfo[i].parts[1] = 0; // установка второго элемента массива равным нулю

threadInfo[i].length = 0; // установка длины массива равной нулю

threadInfo[i].limit = threadInfo[i - 1].parts[0]; // установка границы разбиений, обрабатываемых последним потоком

}

int main()

{

totalCount = 0; // инициализация переменной для подсчета общего количества разбиений

int number = 0; // исходное число, которое требуется разбить

int threadCount = 0; // количество потоков, которое будет использовано для разбиения числа

readFile(&threadCount, &number); // чтение входных данных из файла

int extraThread = threadCount; // запоминаем исходное количество потоков для записи в выходной файл

if (threadCount > number) // если число потоков больше исходного числа, то уменьшаем количество потоков до исходного числа

{

threadCount = number;

}

sem\_init(&semaphore, 0, 1); // инициализация семафора

pthread\_t threads[100]; // массив идентификаторов потоков

int interval = number / threadCount; // длина интервала, на которые разбивается исходное число

divideData(interval, threadCount, number); // разбиение исходного числа на интервалы и подготовка данных для потоков

for (int i = 0; i < threadCount; i++) // создание и запуск потоков

{

pthread\_create(&threads[i], 0, threadFunction, (void \*)((char \*)0 + i));

}

int startTime = clock(); // сохранение времени начала работы программы

int time = startTime;

for (int i = 0; i < threadCount; i++) // ожидание завершения работы потоков

{

pthread\_join(threads[i], 0);

}

startTime = clock() - startTime; // вычисление времени работы программы

startTime /= 1000; // перевод времени в секунды

totalCount -= 1; // уменьшение общего количества разбиений на единицу

FILE \*outputFile = fopen("output.txt", "w"); // открытие выходного файла для записи

FILE \*timeFile = fopen("time.txt", "w"); // открытие файла для записи времени работы программы

fprintf(outputFile, "%d\n", extraThread);

fprintf(outputFile, "%d\n", number); // запись исходного числа в выходной файл

fprintf(outputFile, "%d", totalCount); // запись общего количества разбиений в выходной файл

fprintf(timeFile, "%d", startTime); // запись времени работы программы в файл

fclose(outputFile); // закрытие выходного файла

fclose(timeFile); // закрытие файла времени работы программы

sem\_destroy(&semaphore); // удаление семафора

for (int i = 0; i < threadCount; i++) // освобождение выделенной памяти

{

free((void \*)threadInfo[i].parts);

}

return 0;

}

1. *Листинг кода qsort.cpp*

//

//Описание работы программы :

//-Считывание количества потоков и размера массива из файла input.txt.

//- Считывание самого массива из файла input.txt.

//- Создание первоначальной задачи для сортировки всего массива и добавление ее в очередь задач.

//- Создание указанного количества потоков и запуск каждого потока на выполнение функции thread\_handler, передавая ей ссылку на массив arr.

//- Каждый поток в цикле извлекает задачи из очереди и выполняет их.

//- Если количество элементов в подмассиве меньше 1000, поток сортирует его целиком.

//- Если количество элементов в подмассиве больше или равно 1000, поток разделяет его на две подзадачи и добавляет их в очередь.

//- Если очередь задач пуста и выполняющихся задач нет, поток завершает работу.

//- После завершения работы всех потоков, отсортированный массив записывается в файл output.txt, а время выполнения программы записывается в файл time.txt.

#include <iostream>

#include <fstream>

#include <vector>

#include <queue>

#include <mutex>

#include <condition\_variable>

#include <time.h>

using namespace std;

struct Task {

int left; // Левая граница подмассива.

int right; // Правая граница подмассива.

};

queue<Task> task\_queue; // Очередь задач.

mutex queue\_mutex; // Мьютекс для доступа к очереди.

condition\_variable queue\_cv; // Условная переменная для оповещения потоков о доступности задач.

int active\_tasks = 0; // Количество выполняющихся задач. Счетчик

bool work\_done = false; // Флаг, обозначающий, что задачи выполнены.

// Функция разделения подмассива на две части.

int partition(vector<int>& arr, int low, int high) {

int pivot = arr[high]; // Выбор опорного элемента.

int i = low - 1;

for (int j = low; j <= high - 1; j++) {

if (arr[j] <= pivot) { // Если текущий элемент меньше или равен опорному, меняем его местами с элементом слева от опорного.

i++;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]); // Перемещаем опорный элемент на место разделителя.

return i + 1; // Возвращаем позицию разделителя.

}

// Функция быстрой сортировки.

void quicksort(vector<int>& arr, int low, int high) {

if (low < high) {

int pivot = partition(arr, low, high); // Разделение подмассива на две части.

quicksort(arr, low, pivot - 1); // Сортировка левой части.

quicksort(arr, pivot + 1, high); // Сортировка правой части.

}

}

// Функция, обрабатывающая задачи.

void thread\_handler(vector<int>& arr) {

while (true) {

unique\_lock<mutex> lock(queue\_mutex);

while (task\_queue.empty() && !work\_done) { // Если очередь задач пуста и еще есть работа, поток засыпает.

queue\_cv.wait(lock);

}

if (task\_queue.empty() && work\_done) { // Если очередь задач пуста и работа закончена, поток выходит из цикла.

break;

}

Task task = task\_queue.front(); // Получаем задачу из очереди.

task\_queue.pop(); // Удаляем задачу из очереди.

active\_tasks++; // Увеличиваем количество выполняющихся задач.

lock.unlock(); // Разблокируем мьютекс.

// Если количество элементов в подмассиве меньше 1000, сортируем его целиком.

if (task.right - task.left < 1000) {

quicksort(arr, task.left, task.right);

}

else { // Если количество элементов в подмассиве больше или равно 1000, разделяем его на две подзадачи.

int pivot = partition(arr, task.left, task.right); // Разделение подмассива на две части.

Task task1 = { task.left, pivot - 1 }; // Создание первой подзадачи.

Task task2 = { pivot + 1, task.right }; // Создание второй подзадачи.

lock.lock(); // Блокируем мьютекс.

task\_queue.push(task1); // Добавляем первую подзадачу в очередь.

task\_queue.push(task2); // Добавляем вторую подзадачу в очередь.

queue\_cv.notify\_all(); // Оповещаем все потоки, заблокированные на условной переменной, о доступности задач.

lock.unlock(); // Разблокируем мьютекс.

}

active\_tasks--; // Уменьшаем количество выполняющихся задач.

if (task\_queue.empty() && active\_tasks == 0) { // Если очередь задач пуста и выполняющихся задач нет, заканчиваем работу.

work\_done = true; // Устанавливаем флаг, обозначающий, что задачи выполнены.

queue\_cv.notify\_all(); // Оповещаем все потоки, заблокированные на условной переменной, о завершении работы.

}

}

}

int main() {

int thread\_count, n;

ifstream input("input.txt");

input >> thread\_count >> n;

vector<int> arr(n);

for (int i = 0; i < n; i++) {

input >> arr[i];

}

input.close();

Task initial\_task = { 0, n - 1 }; // Создание первоначальной задачи для сортировки всего массива.

{

lock\_guard<mutex> lock(queue\_mutex);

task\_queue.push(initial\_task); // Добавление первоначальной задачи в очередь.

}

clock\_t start = clock();

vector<thread> threads;

for (int i = 0; i < thread\_count; i++) {

threads.push\_back(thread(thread\_handler, ref(arr))); // Создание и запуск потоков для обработки задач.

}

for (thread& t : threads) {

t.join(); // Ожидание завершения всех потоков.

}

clock\_t end = clock();

clock\_t duration = (clock\_t)(end - start) / CLOCKS\_PER\_SEC;

ofstream output("output.txt");

output << thread\_count << endl << n << endl;

for (int i = 0; i < n; i++) {

output << arr[i] << (i == n - 1 ? "\n" : " "); // Запись отсортированного массива в файл.

}

output.close();

ofstream time\_output("time.txt");

time\_output << duration << endl; // Запись времени выполнения в файл.

time\_output.close();

return 0;

}

1. *Листинг кода msort.cpp*

// подключаем необходимые библиотеки

#include <iostream>

#include <fstream>

#include <vector>

#include <queue>

#include <mutex>

#include <condition\_variable>

#include <thread>

#include <algorithm>

#include <time.h>

using namespace std;

// определяем задание, которое нужно выполнить

struct Task

{

int start;

int end;

};

// создаем класс пула потоков

class ThreadPool

{

public:

// конструктор, который создает и запускает указанное число потоков

ThreadPool(int num\_threads) : done(false)

{

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

{

threads.emplace\_back(&ThreadPool::worker, this);

}

}

// деструктор, который завершает работу пула потоков

~ThreadPool()

{

{

lock\_guard<mutex> lock(tasks\_mutex);

done = true;

}

tasks\_cv.notify\_all();

for (auto &t : threads)

{

t.join();

}

}

// добавление нового задания в очередь заданий

void add\_task(const Task &task)

{

{

lock\_guard<mutex> lock(tasks\_mutex);

tasks.push(task);

}

tasks\_cv.notify\_one();

}

// ожидание завершения всех заданий в очереди

void wait\_for\_tasks()

{

unique\_lock<mutex> lock(tasks\_mutex);

done\_cv.wait(lock, [this]

{ return tasks.empty(); });

}

private:

vector<thread> threads; // вектор для хранения потоков

queue<Task> tasks; // очередь заданий

mutex tasks\_mutex; // мьютекс для защиты доступа к очереди заданий

condition\_variable tasks\_cv; // условная переменная для сигнализации потокам о наличии нового задания

condition\_variable done\_cv; // условная переменная для ожидания завершения всех заданий

bool done; // флаг, указывающий, что работа пула потоков завершена

// метод, который выполняется каждым потоком

void worker()

{

while (true)

{

Task task;

{

unique\_lock<mutex> lock(tasks\_mutex);

tasks\_cv.wait(lock, [this]

{ return !tasks.empty() || done; });

if (tasks.empty() && done)

{

break;

}

task = tasks.front();

tasks.pop();

}

sort(vec.begin() + task.start, vec.begin() + task.end);

{

lock\_guard<mutex> lock(tasks\_mutex);

if (tasks.empty())

{

done\_cv.notify\_one();

}

}

}

}

public:

vector<int> vec; // вектор для хранения исходных данных

};

int main()

{

// открываем файл с входными данными и считываем их

ifstream fin("input.txt");

int num\_threads, n;

fin >> num\_threads >> n;

// создаем пул потоков и задаем размер вектора для хранения исходных данных

ThreadPool pool(num\_threads);

pool.vec.resize(n);

// считываем исходные данные из файла

for (int i = 0; i < n; ++i)

{

fin >> pool.vec[i];

}

fin.close();

// запоминаем время начала работы алгоритма

clock\_t start\_time = clock();

// разбиваем задачу на более мелкие и добавляем их в очередь заданий пула потоков

int step = max(n / (10 \* num\_threads), 1000000);

for (int i = 0; i < n; i += step)

{

int end = min(i + step, n);

pool.add\_task({i, end});

}

// ожидаем завершения всех заданий

pool.wait\_for\_tasks();

// запоминаем время окончания работы алгоритма

clock\_t end\_time = clock();

// записываем результаты работы алгоритма в файлы

ofstream fout("output.txt");

fout << num\_threads << endl;

fout << n << endl;

for (int i = 0; i < n; ++i)

{

fout << pool.vec[i] << ' ';

}

fout << endl;

fout.close();

ofstream time\_fout("time.txt");

time\_fout << (double)(end\_time - start\_time) << endl;

time\_fout.close();

return 0;

}

1. *Листинг кода phil.cpp*

#include <iostream> // Включение библиотеки ввода-вывода

#include <Windows.h> // Включение библиотеки Windows API для работы с потоками и синхронизацией

#include <ctime> // Включение библиотеки для работы со временем

#include <cstdlib> // Включение библиотеки для работы с функцией atoi

using namespace std; // Использование пространства имен стандартной библиотеки

int Phil, a[] = { 1, 2, 3, 4, 5 }, n = -1; // Объявление переменных: количество философов, массив с номерами философов, индекс текущего философа

HANDLE t[5]; // Массив дескрипторов потоков

unsigned long long Total, Started; // Переменные для хранения общего времени и времени начала

HANDLE semaphore; // Дескриптор семафора

CRITICAL\_SECTION criticalSection; // Объект критической секции

CONDITION\_VARIABLE cond; // Переменная условия (не используется в данной версии кода)

const int sec = 1000; // Константа для конвертации миллисекунд в секунды

const int c1 = 1000000; // Константа для конвертации наносекунд в миллисекунды

const unsigned long long int c2 = 1000ull; // Константа для конвертации секунд в миллисекунды

DWORD WINAPI thread(LPVOID p); // Прототип функции потока

void init(); // Прототип функции инициализации

static unsigned long long ticks\_to\_ms(); // Прототип функции конвертации системных тиков в миллисекунды

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

if (argc != 3) { // Проверка на правильное количество аргументов командной строки

cout << "wrong input" << endl; // Вывод сообщения об ошибке

return -1; // Завершение программы с ошибкой

}

Phil = atoi(argv[2]); // Конвертация второго аргумента командной строки (количество философов) в целое число

Total = atoi(argv[1]); // Конвертация первого аргумента командной строки (общее время) в целое число

init(); // Вызов функции инициализации

for (int i = 0; i < 5; i++) { // Цикл создания потоков

t[i] = CreateThread(NULL, 0, thread, NULL, 0, NULL); // Создание потока и сохранение его дескриптора в массиве

}

WaitForMultipleObjects(5, t, TRUE, INFINITE); // Ожидание завершения всех потоков

CloseHandle(semaphore); // Закрытие дескриптора семафора

DeleteCriticalSection(&criticalSection); // Удаление критической секции

return 0; // Завершение программы с успешным статусом

}

DWORD WINAPI thread(LPVOID p) { // Функция потока

unsigned long long time; // Переменная для хранения времени

do { // Начало основного цикла потока

Sleep(Phil); // Приостановка потока на указанное количество времени

WaitForSingleObject(semaphore, INFINITE); // Захват семафора

EnterCriticalSection(&criticalSection); // Вход в критическую секцию

if (ticks\_to\_ms() >= Total) { // Проверка на достижение общего времени

LeaveCriticalSection(&criticalSection); // Выход из критической секции

ReleaseSemaphore(semaphore, 1, NULL); // Освобождение семафора

break; // Выход из цикла потока

}

n = (n + 1) % 5; // Инкрементация индекса текущего философа с циклическим переходом

time = ticks\_to\_ms(); // Получение текущего времени в миллисекундах

time -= Started; // Вычитание времени начала для получения относительного времени

cout << time << ':' << a[n] << ":T->E" << endl; // Вывод информации о текущем философе и переходе из состояния "думает" в состояние "ест"

cout << time << ':' << a[(n + 2) % 5] << ":T->E" << endl; // Вывод информации о следующем философе и переходе из состояния "думает" в состояние "ест"

Sleep(Phil); // Приостановка потока на указанное количество времени

time = ticks\_to\_ms(); // Получение текущего времени в миллисекундах

time -= Started; // Вычитание времени начала для получения относительного времени

cout << time << ':' << a[n] << ":E->T" << endl; // Вывод информации о текущем философе и переходе из состояния "ест" в состояние "думает"

cout << time << ':' << a[(n + 2) % 5] << ":E->T" << endl; // Вывод информации о следующем философе и переходе из состояния "ест" в состояние "думает"

LeaveCriticalSection(&criticalSection); // Выход из критической секции

ReleaseSemaphore(semaphore, 1, NULL); // Освобождение семафора

} while (1); // Конец основного цикла потока

return 0; // Возврат из функции потока

}

static unsigned long long ticks\_to\_ms() { // Функция конвертации системных тиков в миллисекунды

return GetTickCount64(); // Возвращает количество миллисекунд, прошедших с момента загрузки системы

}

void init() { // Функция инициализации

Started = ticks\_to\_ms(); // Установка времени начала

Total += Started; // Добавление времени начала к общему времени

semaphore = CreateSemaphore(NULL, 1, 1, NULL); // Создание семафора с начальным значением 1 и максимальным значением 1

InitializeCriticalSection(&criticalSection); // Инициализация критической секции

}