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

**ЛАБОРАТОРНАЯ РАБОТА № 3**

**«Механизмы многопоточности»**

1. по дисциплине «Операционные системы»
2. Выполнил
3. студент гр. №4851003/00002 Скрипко И.А.

<*подпись*>

1. Преподаватель
2. Крундышев В.М.

<*подпись*>

1. Санкт-Петербург
2. 2022
3. **Цель работы**

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

1. **Задачи**

Реализовать следующие программы:

1. Вычисления: ОС – Windows, синхронизация – Mutex, Event. Программа должна находить количество уникальных разложений на слагаемые.
2. QuickSort: ОС – Windows, синхронизация – Semaphore, Crit.section
3. MergeSort: ОС – Linux, синхронизация – Mutex, Cond. Variable
4. Философы: ОС – Linux, синхронизация – Semaphore, метод решения – управляющий процесс. Программа должна создать 7 потоков: main, управляющий и 5 философов, которые могут есть и думать, причем есть могут максимум только 2 из 5 одновременно – в этом случае они не должны быть соседями.
5. **Ход работы**
   1. **Алгоритм работы expr.cpp**

Задача решается следующим алгоритмом:

1. Число разбивается на сумму единиц;
2. Далее ищется слева направо самое маленькое слагаемое (среди всех, кроме последнего ненулевого слагаемого);
3. К найденному слагаемому добавляется 1 из последнего ненулевого слагаемого (из последнего ненулевого слагаемого 1 отнимается);
4. Все числа после найденного слагаемого разлагаются на сумму единиц;
5. Увеличить счетчик уникальных разложений на 1;
6. Если первое слагаемое не равно начальному числу, то перейти на шаг 2.

В данном алгоритме можно заметить, что первое число будет 1 изначально, когда оно станет равным 2 (после первой итерации), то после первого слагаемого все числа будут равны единицам. Далее последовательно слагаемые будут превращаться в двойки до тех пор, пока все элементы не станут равны 2, далее слева направо за самый маленький элемент будет принята первая двойка, а следовательно, все остальные разложатся на единицы и получится тройка, после которой идут все единицы. Это свойство будет выполняться каждый раз.

Благодаря этому свойству можно разбить обработку данных на потоки, которые будут обрабатывать данный до тех пор, пока первое число не будет равно установленной границе.

Например, число 10 будет разбито на 2 потока со следующими входными данными (число и количество потоков задается во входном файле, программа сама подбирает границы по входным данным):

1 1 1 1 1 1 1 1 1 1 – обработка до 5 1 1 1 1 1 0 0 0 0;

5 1 1 1 1 1 – обработка до 10 0 0 0 0 0 0 0 0 0.

На рисунке 1 представлена блок-схема алгоритма работы поточной функции программы expr.cpp.

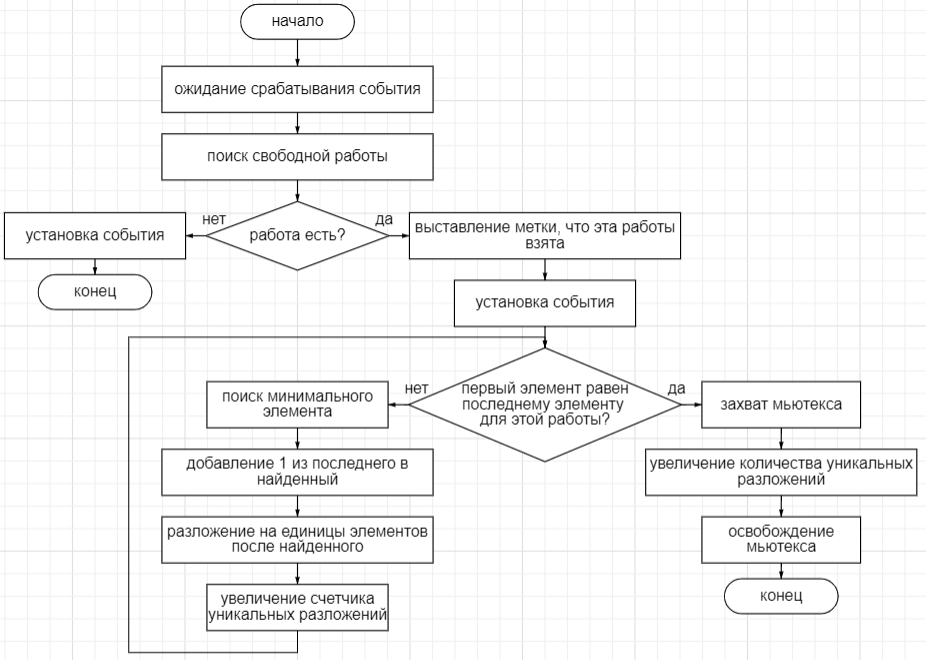


Рисунок 1 – Блок-схема алгоритма работы поточной функции программы expr.cpp

В таблице 1 представлены разработанные функции.

Таблица 1 – Разработанные функции

|  |  |  |  |
| --- | --- | --- | --- |
| *Имя функции* | *Аргументы* | *Возвращаемое значение* | *Описание* |
| DWORD WINAPI expr(void\* param) | Аргумент не используется | Описатель созданной нити | Выбор данных для работы и их обработка, увеличение счетчика уникальных разложений на результат работы. |
| int main() | - | Успех/не успех выполнения | Считывание данных из входного файла, разбиение на массивы работ, создание потоков, вывод результатов в выходной файл. |

Таблица 2 – Временная диаграмма взаимодействия потоков в моменты синхронизации

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Thread/time | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | Ready | Blocked | Blocked | - | - | - | - | - |
| 2 | Ready | Ready | Ready | Blocked | Blocked | - | - | - |
| 3 | Ready | Ready | Ready | Ready | Ready | Blocked | Blocked | - |

Где «Ready» показывает готовность выполняться, «Blocked» – захват элемента синхронизации, «-» – потоку больше не нужен этот элемент синхронизации.

В таблице 3 показаны результаты тестирования программы. Указывается число N, далее в строчку пишется время работы в мс с 1, 2, 3, 4, 5, 8 и 10 потоками

соответственно.

Таблица 3 – Результаты тестирования программы

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 8 | 10 |
| 73 | 890 | 813 | 766 | 563 | 516 | 515 | 422 |
| 80 | 2266 | 2187 | 1969 | 1547 | 1312 | 1469 | 1062 |
| 85 | 4407 | 4343 | 3922 | 3078 | 2469 | 2781 | 2094 |
| 90 | 8547 | 8328 | 7687 | 5969 | 4719 | 5625 | 4329 |
| 93 | 12796 | 12329 | 11453 | 9031 | 6984 | 7578 | 5516 |
| 95 | 16078 | 15750 | 14391 | 10969 | 7906 | 9515 | 6953 |
| 98 | 23359 | 23422 | 20922 | 16469 | 11937 | 14750 | 10188 |
| 100 | 29890 | 29532 | 27218 | 22188 | 15765 | 19500 | 15328 |
| 105 | 54735 | 53750 | 50375 | 40437 | 29078 | 42125 | 37297 |
| 110 | 116359 | 97688 | 91468 | 74422 | 55531 | 66110 | 53765 |

* 1. **Алгоритм работы программы qsort.cpp**

Задача решается следующим алгоритмом:

1. Входной массив разбивается на подмассивы равных размеров;
2. Каждый массив сортируется потоковой функцией;
3. Делается слияние всех массивов в 1.

На рисунке 2 представлена блок-схема алгоритма работы поточной функции программы qsort.cpp.

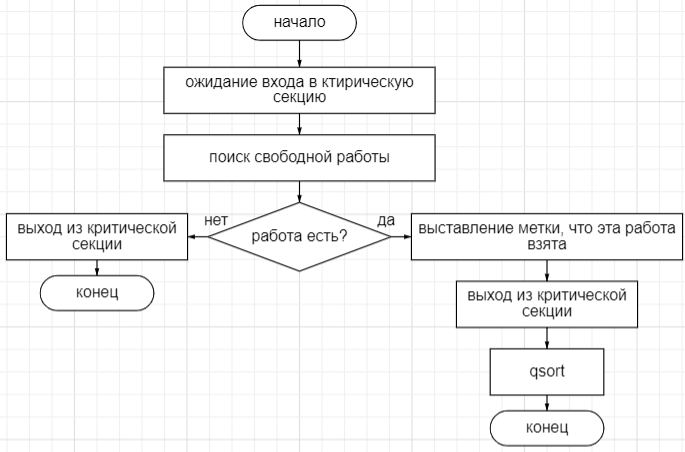


Рисунок 2 – Блок-схема поточной функции программы qsort.cpp

В таблице 4 представлены разработанные функции.

Таблица 4 – Разработанные функции

|  |  |  |  |
| --- | --- | --- | --- |
| *Имя функции* | *Аргументы* | *Возвращаемое значение* | *Описание* |
| DWORD WINAPI expr(void\* param) | Аргумент не используется | Описатель созданной нити | Выбор данных для работы и вызов qsort |
| int main() | - | Успех/не успех выполнения | Считывание данных из входного файла, инициализация элементов синхронизации, разбиение входного массива на подмассивы, создание потоков, вывод результатов в выходной файл. |
| void q\_sort(int l, int r); | Границы массива | - | Выполняет qsort |
| int merge(); | - | Успех/не успех | Выполняет merge над T отсортированными подмассивами |

В таблице 5 показаны результаты тестирования программы. Указывается число N, далее в строчку пишется время работы в мс с 1, 2, 3, 4, 5, 8 и 10 потоками соответственно.

Таблица 5 – Время работы программы с различными входными данными

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 8 | 10 |
| 1000000 | 156 | 94 | 63 | 63 | 47 | 62 | 78 |
| 2000000 | 328 | 188 | 156 | 125 | 125 | 125 | 172 |
| 3000000 | 484 | 281 | 234 | 203 | 203 | 187 | 234 |
| 4000000 | 657 | 391 | 297 | 266 | 266 | 281 | 313 |
| 5000000 | 844 | 515 | 391 | 344 | 344 | 328 | 391 |
| 6000000 | 1000 | 578 | 468 | 422 | 407 | 406 | 453 |
| 7000000 | 1188 | 672 | 531 | 500 | 453 | 469 | 516 |
| 8000000 | 1344 | 781 | 625 | 547 | 532 | 531 | 609 |
| 9000000 | 1531 | 875 | 734 | 641 | 578 | 593 | 671 |
| 10000000 | 1719 | 984 | 781 | 703 | 656 | 672 | 750 |

* 1. **Алгоритм работы программы msort.cpp**

Задача решается следующим алгоритмом:

1. Входной массив разбивается на подмассивы равных размеров;
2. Каждый массив сортируется потоковой функцией;
3. Делается слияние всех массивов в 1.

На рисунке 3 представлена блок-схема алгоритма работы поточной функции программы msort.cpp.

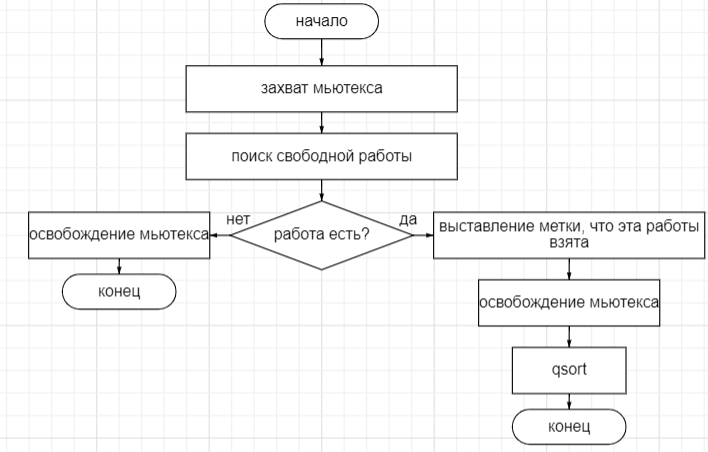


Рисунок 3 – Блок-схема поточной функции программы msort.cpp

В таблице 6 представлены разработанные функции.

Таблица 6 – Разработанные функции

|  |  |  |  |
| --- | --- | --- | --- |
| *Имя функции* | *Аргументы* | *Возвращаемое значение* | *Описание* |
| void\* thread\_entry(void\* param); | Аргумент не используется | Успех/не успех создания | Выбор данных для работы и вызов msort |
| int main() | - | Успех/не успех выполнения | Считывание данных из входного файла, инициализация элементов синхронизации, разбиение входного массива на подмассивы, создание потоков, вывод результатов в выходной файл. |
| unsigned long to\_ms(struct timespec\* tm); | Структура, в которой хранится время | Время в мс | Переводит формат структуры в мс |
| int merge(); | - | Успех/не успех | Выполняет merge над T отсортированными подмассивами |
| void Merge(int first, int last); | Границы массива | - | Выполняет слияние двух отсортированных массивов |
| void MergeSort(int first, int last); | Границы массива | - | Выполняет MergeSort |

В таблице 7 показаны результаты тестирования программы. Указывается число N, далее в строчку пишется время работы в мс с 1, 2, 3, 4, 5, 8 и 10 потоками соответственно.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 8 | 10 |
| 1000000 | 300 | 287 | 276 | 311 | 259 | 195 | 255 |
| 2000000 | 829 | 593 | 619 | 703 | 561 | 517 | 486 |
| 3000000 | 879 | 540 | 511 | 490 | 504 | 530 | 557 |
| 4000000 | 1179 | 646 | 647 | 659 | 671 | 701 | 729 |
| 5000000 | 1485 | 809 | 821 | 832 | 895 | 878 | 909 |
| 6000000 | 1775 | 950 | 1025 | 991 | 990 | 1041 | 1074 |
| 7000000 | 2084 | 1119 | 1139 | 1164 | 1169 | 1219 | 1265 |
| 8000000 | 2411 | 1325 | 1312 | 1325 | 1334 | 1399 | 1448 |

* 1. **Алгоритм работы phil.cpp**

Задача решается следующим алгоритмом:

1. Выстраивается очередь философов для перехода в состояние Е;
2. Каждому философу соответствует семафор, который инициализируется 0;
3. Управляющий поток берет из очереди номера 2 потоков и поднимает им семафоры, далее спит в течение T миллисекунд (столько философы будут есть), после чего опускает их снова;
4. Если очередь закончилась, то конец, иначе переход на пункт 3.

На рисунках 4-5 представлены блок-схемы алгоритмов работы поточных функций программы phil.cpp.

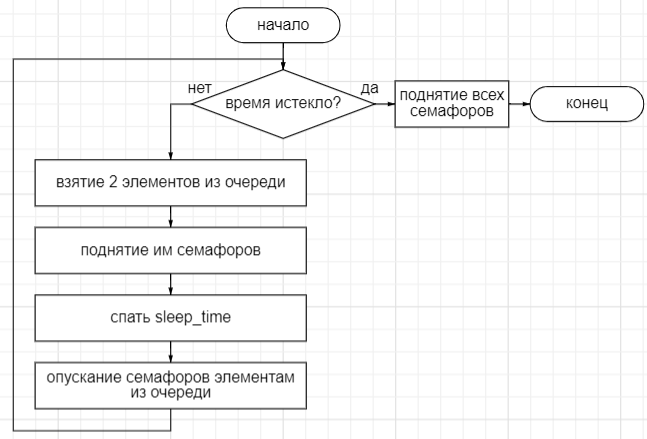


Рисунок 4 – Блок-схема поточной функции управляющего потока

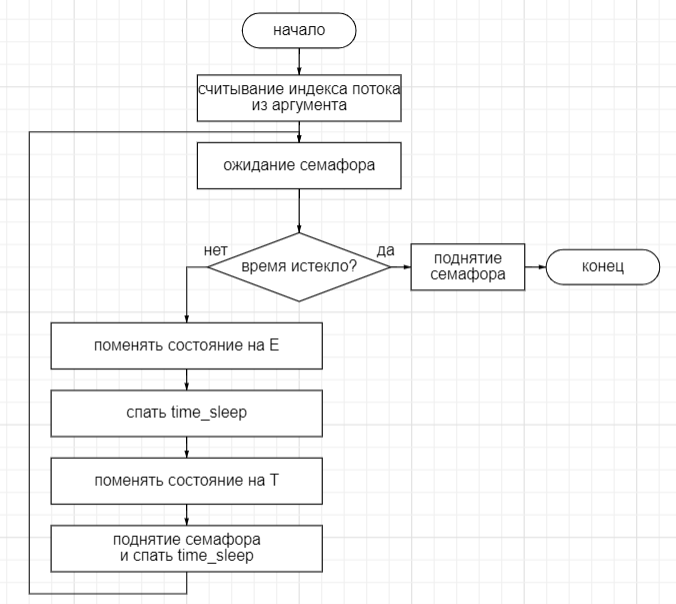


Рисунок 5 – Блок-схема поточной функции философа

В таблице 8 представлены разработанные функции.

Таблица 8 – Разработанные функции

|  |  |  |  |
| --- | --- | --- | --- |
| *Имя функции* | *Аргументы* | *Возвращаемое значение* | *Описание* |
| int main() | - | Успех/не успех выполнения | Засечение времени начала работы функции. Вызов threads\_init и threads\_deinit. |
| unsigned long long get\_time(void); | - |  |  |
| void threads\_init(int, char\*\*); | Аргументы командной строки | - | Создание очереди взятия вилок, инициализация семафоров, создание потоков, ожидание потоков |
| void threads\_deinit(void); |  | - | Деинициализация созданных элементов и освобождение памяти |
| void\* servant\_thread\_func(void\*); | Не используется | Успех/не успех создания | Управление философами |
| void\* phil\_thread\_func(void\*); | Номер философа | Успех/не успех создания | Философы |

В таблице 9 представлены результаты тестирования программы.

Таблица 9 – Результаты тестирования программы

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Философы | Phil1 | | Phil2 | | Phil3 | | Phil4 | | Phil5 | | | |
| TOTAL/PHIL | E | T | E | T | E | T | E | T | E | | T | |
| 200/10 | 70 | 138 | 80 | 128 | 70 | 138 | 80 | 128 | 80 | | 128 | |
| 200/20 | 80 | 126 | 80 | 126 | 80 | 126 | 80 | 126 | 80 | | 126 | |
| 200/50 | 100 | 103 | 50 | 153 | 100 | 103 | 50 | 153 | 100 | | 103 | |
| 400/40 | 160 | 248 | 160 | 248 | 160 | 248 | 160 | 248 | 160 | | 248 | |
| 400/80 | 160 | 245 | 160 | 245 | 160 | 245 | 160 | 245 | 160 | | 245 | |
| 400/100 | 200 | 204 | 100 | 304 | 200 | 204 | 100 | 304 | 200 | | 204 | |
| 1000/100 | 400 | 611 | 400 | 611 | 400 | 611 | 400 | 611 | 400 | | 611 | |
| 1000/200 | 400 | 606 | 400 | 606 | 400 | 606 | 400 | 606 | 400 | | 606 | |
| 1000/400 | 400 | 803 | 400 | 803 | 400 | 803 | 0 | 1203 | 400 | | 803 | |
| 2000/200 | 800 | 1212 | 800 | 1212 | 800 | 1212 | 800 | 1212 | 800 | | 1212 | |
| 2000/400 | 800 | 1206 | 800 | 1206 | 800 | 1206 | 800 | 1206 | | 800 | | 1206 | |
| 2000/800 | 800 | 1602 | 800 | 1602 | 800 | 1602 | 0 | 2202 | | 800 | | 1602 | |
| 4000/400 | 1600 | 2411 | 1600 | 2411 | 1600 | 2411 | 1600 | 2411 | | 1600 | | 2411 | |
| 4000/800 | 1600 | 2406 | 1600 | 2406 | 1600 | 2406 | 1600 | 2406 | | 1600 | | 2406 | |
| 4000/1000 | 2000 | 2004 | 1000 | 3004 | 2000 | 2004 | 1000 | 3004 | | 2000 | | 2004 | |
| 20000/2000 | 8000 | 12013 | 8000 | 12013 | 8000 | 12013 | 8000 | 12013 | | 8000 | | 12013 | |
| 20000/4000 | 8000 | 12006 | 8000 | 12006 | 8000 | 12006 | 8000 | 12006 | | 8000 | | 12006 | |
| 20000/8000 | 8000 | 16002 | 8000 | 16002 | 8000 | 16002 | 0 | 24002 | | 8000 | | 16002 | |

1. **Выводы**

В ходе выполнения данной лабораторной работы были изучены принципы разработки многопоточных программ, изучены программные интерфейсы операционных систем windows и linux для организации многопоточности, получены навыки организации взаимодействия потоков в многопоточных программах.

В программе expr.cpp видно, что при увеличении количества потоков время работы уменьшается, кроме 8 потоков – возможно при делении промежутки получаются примерно одинаковые по времени выполнения и попадают на синхронизацию неудачным образом (в программе присутствуют 2 места синхронизации), из-за чего время падает.

В программе qsort.cpp при запуске на 10 потоках время работы ухудшается – возможно это связано с последним действием – слиянием. Массивы отсортируются меньшими размерами, что увеличит скорость, но далее будет происходить больше сравнений из-за слияния (сортировка слиянием имеет сложность худшую относительно быстрой сортировки), что увеличит время работы программы.

В программе msort.cpp можно наблюдать ту же проблему, что и в qsort, но не всегда и с меньшим вредом – это можно объяснить тем, что программа сортируется примерно так же, как и делается последнее действие, поэтому сложность работы одна и та же.

В программе phil.cpp при малом времени PHIL происходит много переключений и использования средств синхронизации, что добавляет времени работы, но каждый поток работает примерно одинаковое время. При больших PHIL происходит мало переключений и дополнительно тратится мало времени, но может получиться так, что какому-то потоку достанется намного меньше времени, чем другим (или вообще не достанется).

**ПРИЛОЖЕНИЕ А**

**Листинг программы expr.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <windows.h>//средства синхронизации

#include <stdio.h>

HANDLE\* t;//описатели потоков

HANDLE mutex;//мьютекс для синхронизации потоков (доступ к общему количеству разложений)

volatile int cnt=0;//общее количество разложений

//volatile, чтобы не было добавлено в регистр, а все время работа была с памятью

int\* vars;

int\*\* arrays\_for\_ts;

int\* threads;

int T, N;

HANDLE event; // Событие для синхронизации доступа к массиву с данными

//поточная функция, param - номер потока и индекс для массива, с которым будет работа

DWORD WINAPI expr(void\* param)

{

// Ожидание срабатывания события

WaitForSingleObject(event, INFINITE);

int ind=-1;

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

if (vars[i] != -1)

{

ind = vars[i];

vars[i] = -1;

break;

}

if (ind == -1)

{

SetEvent(event);

return 0;

}

//printf("%d\n",ind);

int size = N - threads[ind] + 1;//размер массива

int\* local\_array = (int\*)malloc(size \* sizeof(int));

memcpy(local\_array,arrays\_for\_ts[ind],size\*sizeof(int));

// Установка события - теперь со списком vars может

// поработать другой поток, а в текущем этого делать больше нельзя

SetEvent(event);

int local\_cnt = 0;

while (1)

{

//printf("%d\n", array[0]);

/\*printf("size=%d\n", size);;

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

printf("%d ", array[i]);

printf("\n");\*/

//если поток закончил свою часть работы

if (ind == T - 1)

if (local\_array[0] == N) break;

else;

else

if (local\_array[0] == threads[ind + 1]) break;

/\*

Элемент в нулевом индексе нашего динамического

массива на текущий момент.

\*/

int first\_elem = local\_array[0];

int i = 0, min\_elem=0;

//рассматриваем все, кроме последнего, последний отдаст 1 минимальному

while (i < size-1)

{

/\*Найдем элемент меньше первого.\*/

if (local\_array[i] < first\_elem)

{

first\_elem = local\_array[i];

min\_elem = i;

}

i++;

}

//printf("size=%d\n", size);;

//for (int i = 0; i < size; i++)

//printf("%d ", local\_array[i]);

//printf("\n");

//printf("min\_elem=%d\n", min\_elem);

/\*Перенос 1 из последнего элемента в минимальный\*/

local\_array[min\_elem] += 1;

local\_array[size-1] -= 1;

/\*Обрежем массив, если это надо, и найдем сумму его элементов, после элемента, который увеличили.\*/

if (local\_array[size - 1] == 0) size--;

int array\_sum = 0;

for (int i = min\_elem + 1; i < size; i++)

array\_sum += local\_array[i];

//printf("new\_size=%d\n", size);

//for (int i = 0; i < size; i++)

//printf("%d ", local\_array[i]);

//printf("\n");

//printf("array\_sum=%d\n",array\_sum);

/\*Добавим в массив единицы заново с учетом суммы.\*/

for (int j = 0; j < array\_sum; j++)

{

local\_array[min\_elem + 1 + j] = 1;

//printf("array[%d]=%d ", min\_elem + 1 + j, local\_array[min\_elem + 1 + j]);

}

//printf("\n");

//найдем новый размер массива

size = 0;

for (int j = 0;; j++)

if (local\_array[j] != 0)

++size;

else break;

local\_cnt++;

}

//увеличение счетчика на найденное количество разложений

WaitForSingleObject(mutex, INFINITE);

//printf("[%d] cnt=%d\n", ind, local\_cnt);

cnt += local\_cnt;

ReleaseMutex(mutex);

free(local\_array);

return 0;

}

int main()

{

int masssss[10] = { 73, 80, 85, 90, 93, 95, 98, 100, 105, 110 };

int masss[7] = { 1,2,3,4,5,8,10 };

for (int iiiii = 0; iiiii < 10; iiiii++)

{

printf("\nN=%d:\n", masssss[iiiii]);

for (int iiii = 0; iiii < 7; iiii++)

{

T = masss[iiii];

N = masssss[iiiii];

DWORD finished, started;

mutex = CreateMutex(NULL, FALSE, NULL);

// Создание события: с автосбросом, изначально в сигнальном состоянии

event = CreateEvent(NULL, FALSE, TRUE, NULL);

/\*FILE\* input = fopen("input.txt", "r");

if (input == NULL)

{

printf("error: fopen\n");

return -1;

}

fscanf(input, "%d%d", &T, &N);

fclose(input);\*/

threads = (int\*)malloc(T \* sizeof(int));

if (threads == NULL)

{

printf("error: malloc\n");

return -1;

}

memset(threads, 0, T \* sizeof(int));

vars = (int\*)malloc(T \* sizeof(int));

if (vars == NULL)

{

printf("error: malloc\n");

return -1;

}

memset(vars, 0, T \* sizeof(int));

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

vars[i] = i;

//printf("%d %d\n", vars[0], vars[1]);

arrays\_for\_ts = (int\*\*)malloc(T \* sizeof(int\*));//массив с указателями на массивы

if (arrays\_for\_ts == NULL)

{

printf("error: malloc\n");

free(threads);

return -1;

}

memset(arrays\_for\_ts, NULL, T \* sizeof(int\*));

t = (HANDLE\*)malloc(T \* sizeof(HANDLE));

if (t == NULL)

{

printf("error: malloc\n");

free(threads);

free(arrays\_for\_ts);

return -1;

}

memset(t, 0, T \* sizeof(HANDLE));

/\*

распределение нагрузки. Например, для входных значение 3 16:

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

5 1 1 1 1 1 1 1 1 1 1 1

10 1 1 1 1 1 1

массивы для каждого потока

\*/

threads[0] = 1;

arrays\_for\_ts[0] = (int\*)malloc(N \* sizeof(int));

if (arrays\_for\_ts[0] == NULL)

{

printf("error: malloc\n");

free(threads);

free(arrays\_for\_ts);

free(t);

return -1;

}

memset(arrays\_for\_ts[0], 0, N \* sizeof(int));

for (int j = 0; j < N; j++)

arrays\_for\_ts[0][j] = 1;

/\*for (int j = 0; j < N; j++)

printf("%d ",arrays\_for\_ts[0][j]);\*/

for (int i = 1; i < T; i++)

{

threads[i] = N / T \* i;//первые элементы в рабочих массивах

if (threads[i] == 0)

{

vars[i] = -1;

threads[i] = N;

}

arrays\_for\_ts[i] = (int\*)malloc((N - threads[i] + 1) \* sizeof(int));//размеры рабочих массивов

if (arrays\_for\_ts[i] == NULL)

{

printf("error: malloc\n");

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

free(arrays\_for\_ts[j]);

free(threads);

free(arrays\_for\_ts);

free(t);

return -1;

}

memset(arrays\_for\_ts[i], 0, (N - threads[i] + 1) \* sizeof(int));

arrays\_for\_ts[i][0] = threads[i];//первый элемент принимает значение большое

for (int j = 1; j < N - threads[i] + 1; j++)

arrays\_for\_ts[i][j] = 1;

}

//создание всех потоков

//printf("%d\n", T);

int i = 0;

for (; i < T; i++)

{

//printf("%d ", i);

t[i] = CreateThread(0, 0, expr, 0, 0, 0);

if (0 == t[i])

{

printf("CreateThread failed. GetLastError(): %u\n", GetLastError());

return -1;

}

}

started = GetTickCount(); // Получение времени (в мс)

// Ожидание завершения потоков

WaitForMultipleObjects(T, t, TRUE, INFINITE);

/\*WaitForSingleObject(mutex, INFINITE);

cnt+=T;//каждый раз не учитывался первый заход

ReleaseMutex(mutex);\*/

finished = GetTickCount();

FILE\* output;

output = fopen("output.txt", "w");

if (output == NULL)

{

printf("error: fopen\n");

return -1;

}

fprintf(output, "%d\n%d\n%d", T, N, cnt);

fclose(output);

output = fopen("time.txt", "w");

if (output == NULL)

{

printf("error: fopen\n");

return -1;

}

fprintf(output, "%d", finished - started);

fclose(output);

//printf("Thread complete. Execution time: %u ms\ncnt=%d\n", finished - started, cnt);

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

CloseHandle(t[i]);

CloseHandle(mutex);

CloseHandle(event);

for (int j = 0; j < T; j++)

free(arrays\_for\_ts[j]);

free(threads);

free(arrays\_for\_ts);

free(t);

fprintf(stdout, "%d ", finished - started);

}

}

return 0;

}

**ПРИЛОЖЕНИЕ Б**

**Листинг программы qsort.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <windows.h>

#include <stdio.h>

typedef struct parameters

{

int l, r, flag;

} param\_t;

HANDLE\* t;//описатели потоков

CRITICAL\_SECTION cs;

int \*arrayqs;//массив для Быстрой Сортировки

long long T, N;

param\_t\* input\_data;

void q\_sort(int l, int r);

int merge();

DWORD WINAPI expr(void\* param);

int main()

{

long long masssss[10] = { 1000000, 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000, 10000000};

int masss[7] = { 1,2,3,4,5,8,10 };

for (int iiiii = 0; iiiii < 10; iiiii++)

{

printf("\nN=%lld:\n", masssss[iiiii]);

for (int iiii = 0; iiii < 7; iiii++)

{

DWORD finished, started;

InitializeCriticalSection(&cs);

FILE\* input = fopen("input.txt", "r");

if (input == NULL)

{

printf("error: fopen\n");

return -1;

}

fscanf(input, "%d%d", &T, &N);

T = masss[iiii];

N = masssss[iiiii];

arrayqs = (int\*)malloc(N \* sizeof(int));

if (arrayqs == NULL)

{

printf("error: malloc\n");

DeleteCriticalSection(&cs);

return -1;

}

memset(arrayqs, 0, N \* sizeof(int));

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

fscanf(input, "%d", &arrayqs[i]);

fclose(input);

t = (HANDLE\*)malloc(T \* sizeof(HANDLE));

if (t == NULL)

{

printf("error: malloc\n");

free(arrayqs);

DeleteCriticalSection(&cs);

return -1;

}

memset(t, 0, T \* sizeof(HANDLE));

//распределение нагрузки

//определение количества кусков, на которые будет разбит массив

//выделение памяти под массив с данными для обработки (отсюда потоки будут брать задачи)

input\_data = (param\_t\*)malloc(T \* sizeof(param\_t));

if (input\_data == NULL)

{

printf("error: malloc\n");

free(arrayqs);

free(t);

DeleteCriticalSection(&cs);

return -1;

}

memset(input\_data, 0, T \* sizeof(param\_t));

//расставление границ задачам

if (N < 1000)//если мало элементов, то считать будет один поток

{

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

{

input\_data[i].flag = -1;//не доступно для обработки

input\_data[i].l = -1;

input\_data[i].r = -1;

}

input\_data[0].flag = 1;

input\_data[0].l = 0;

input\_data[0].r = N - 1;

}

else

{

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

{

input\_data[i].l = N / T \* i + 1;

input\_data[i].r = N / T \* (i + 1);

input\_data[i].flag = 1;//доступно для обработки

}

input\_data[T - 1].r = N - 1;

input\_data[0].l = 0;

}

/\*for (int i = 0; i < T; i++)

{

printf("i=%d. l=%d ; r=%d ;\n", i, input\_data[i].l, input\_data[i].r);

}\*/

int i = 0;

for (; i < T; i++)

{

//printf("%d ", i);

t[i] = CreateThread(0, 0, expr, 0, 0, 0);

if (0 == t[i])

{

printf("CreateThread failed. GetLastError(): %u\n", GetLastError());

return -1;

}

}

started = GetTickCount(); // Получение времени (в мс)

WaitForMultipleObjects(T, t, TRUE, INFINITE);

/\*for (int i = 0; i < N; i++)

if (i != N - 1) fprintf(stdout, "%d ", arrayqs[i]);

else fprintf(stdout, "%d", arrayqs[i]);\*/

//все отсиртировано покусочно, осталось сделать merge

merge();

finished = GetTickCount();

FILE\* output;

output = fopen("output.txt", "w");

if (output == NULL)

{

printf("error: fopen\n");

return -1;

}

fprintf(output, "%d\n%d\n", T, N);

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

if (i != N - 1) fprintf(output, "%d ", arrayqs[i]);

else fprintf(output, "%d", arrayqs[i]);

fclose(output);

output = fopen("time.txt", "w");

if (output == NULL)

{

printf("error: fopen\n");

return -1;

}

fprintf(output, "%d", finished - started);

fclose(output);

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

CloseHandle(t[i]);

DeleteCriticalSection(&cs);

free(t);

free(input\_data);

free(arrayqs);

fprintf(stdout, "%d ", finished - started);

}

}

return 0;

}

DWORD WINAPI expr(void\* param)

{

//берем работу из массива с работами

EnterCriticalSection(&cs);

int ind = -1;

int l = 0, r = 0;

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

if (input\_data[i].flag != -1)

{

ind = i;

l = input\_data[i].l;

r = input\_data[i].r;

input\_data[i].flag = -1;

break;

}

if (ind == -1)

{

LeaveCriticalSection(&cs);

return 0;

}

LeaveCriticalSection(&cs);

q\_sort(l, r);

return 0;

}

void q\_sort(int l, int r)

{

int i, j;//индексы

int b;//значение середины

int temp;//для замены

i = l; j = r;//i -левая граница рассматриваемой части массива, j -правая граница рассматриваемой части массива

b = arrayqs[(i + j) / 2];//b -значение середины рассматриваемой части массива

do

{

while (arrayqs[i] < b)

{//ищем неправильно стоящий элемент, начиная с левой части рассматриваемой части массива

i++;

}

while (arrayqs[j] > b) {//ищем неверно стоящий элемент, начиная с правой границы рассматриваемой части массива

j--;

}

if (i <= j)

{//если индексы у найденных элементов верные, то меняются местами

temp = arrayqs[i];

arrayqs[i] = arrayqs[j];

arrayqs[j] = temp;

i++;

j--;

}

}

while (i <= j);//блок повторяется до тех пор, пока слева от элемента, взятого за середину, не будут все элементы меньше, а справа больше его

if (j > l)

q\_sort(l,j);//рекурсивно вызывается эта функция для правой от середины части и левой, уже с новыми границами

if (i < r)

q\_sort(i,r);

}

int merge()

{

int\* lefts = (int\*)malloc(T \* sizeof(int));

if (lefts == NULL)

{

printf("error: malloc\n");

return -1;

}

int\* rights = (int\*)malloc(T \* sizeof(int));

if (rights == NULL)

{

free(lefts);

printf("error: malloc\n");

return -1;

}

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

{

lefts[i]=input\_data[i].l;

rights[i]=input\_data[i].r;

}

int\* data\_cpy = (int\*)malloc(N \* sizeof(int));

if (data\_cpy == NULL)

{

free(lefts);

free(rights);

printf("error: malloc\n");

return -1;

}

memcpy(data\_cpy, arrayqs, N\*sizeof(int));

/\*printf("\n");

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

{

printf("i=%d. l=%d ; r=%d ;\n",i,lefts[i],rights[i]);

}

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

if (i != N - 1) fprintf(stdout, "%d ", data\_cpy[i]);

else fprintf(stdout, "%d", data\_cpy[i]);\*/

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

{

int min=0, min\_ind=0;

//выбираю минимальный элемент для старта

for (int j = 0; j < T; j++)

{

if (lefts[j] <= rights[j]&& lefts[j]!=-1)

{

min = data\_cpy[lefts[j]];

min\_ind = j;

break;

}

}

//выбираю среди T массивов минимальный элемент среди левых

for (int j = min\_ind+1; j < T; j++)

{

if (lefts[j]<=rights[j]&&min > data\_cpy[lefts[j]]&& lefts[j]!=-1)

{

min = data\_cpy[lefts[j]];

min\_ind = j;

}

}

//ставлю элемент в итоговый массив

arrayqs[i] = min;

lefts[min\_ind]++;

}

free(lefts);

free(rights);

free(data\_cpy);

return 0;

}

**ПРИЛОЖЕНИЕ В**

**Листинг программы msort.cpp**

#include <pthread.h>

#include <malloc.h>

#include <stdio.h>

#include <string.h>

#include <cerrno>

typedef struct parameters

{

int l, r, flag;

} param\_t;

int\* arrayms;

int T, N;

param\_t\* input\_data;

pthread\_mutex\_t mutex;

unsigned long to\_ms(struct timespec\* tm);

void\* thread\_entry(void\* param);

void Merge(int first, int last);

void MergeSort(int first, int last);

int merge();

unsigned long to\_ms(struct timespec\* tm)

{

return ((unsigned long) tm->tv\_sec \* 1000 +

(unsigned long) tm->tv\_nsec / 1000000);

}

void\* thread\_entry(void\* param)

{

//берем работу из массива с работами

pthread\_mutex\_lock(&mutex);

int ind = -1;

int l = 0, r = 0;

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

if (input\_data[i].flag != -1)

{

ind = i;

l = input\_data[i].l;

r = input\_data[i].r;

input\_data[i].flag = -1;

break;

}

if (ind == -1)

{

pthread\_mutex\_unlock(&mutex);

return 0;

}

pthread\_mutex\_unlock(&mutex);

MergeSort(l, r);

return 0;

}

//функция, сливающая массивы

void Merge(int first, int last)

{

int middle, start, final, j;

int\* mas = (int\*)malloc(N\*sizeof(int));

middle = (first + last) / 2; //вычисление среднего элемента

start = first; //начало левой части

final = middle + 1; //начало правой части

for (j = first; j <= last; j++) //выполнять от начала до конца

if ((start <= middle) && ((final > last) || (arrayms[start] < arrayms[final])))

{

mas[j] = arrayms[start];

start++;

}

else

{

mas[j] = arrayms[final];

final++;

}

//возвращение результата в список

for (j = first; j <= last; j++) arrayms[j] = mas[j];

free(mas);

};

//рекурсивная процедура сортировки

void MergeSort(int first, int last)

{

{

if (first < last)

{

MergeSort(first, (first + last) / 2); //сортировка левой части

MergeSort((first + last) / 2 + 1, last); //сортировка правой части

Merge(first, last); //слияние двух частей

}

}

};

int merge()

{

int\* lefts = (int\*)malloc(T \* sizeof(int));

if (lefts == NULL)

{

printf("error: malloc\n");

return -1;

}

int\* rights = (int\*)malloc(T \* sizeof(int));

if (rights == NULL)

{

free(lefts);

printf("error: malloc\n");

return -1;

}

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

{

lefts[i] = input\_data[i].l;

rights[i] = input\_data[i].r;

}

int\* data\_cpy = (int\*)malloc(N \* sizeof(int));

if (data\_cpy == NULL)

{

free(lefts);

free(rights);

printf("error: malloc\n");

return -1;

}

memcpy(data\_cpy, arrayms, N \* sizeof(int));

/\*printf("\n");

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

{

printf("i=%d. l=%d ; r=%d ;\n",i,lefts[i],rights[i]);

}

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

if (i != N - 1) fprintf(stdout, "%d ", data\_cpy[i]);

else fprintf(stdout, "%d", data\_cpy[i]);\*/

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

{

int min = 0, min\_ind = 0;

//выбираю минимальный элемент для старта

for (int j = 0; j < T; j++)

{

if (lefts[j] <= rights[j] && lefts[j] != -1)

{

min = data\_cpy[lefts[j]];

min\_ind = j;

break;

}

}

//выбираю среди T массивов минимальный элемент среди левых

for (int j = min\_ind + 1; j < T; j++)

{

if (lefts[j] <= rights[j] && min > data\_cpy[lefts[j]] && lefts[j] != -1)

{

min = data\_cpy[lefts[j]];

min\_ind = j;

}

}

//ставлю элемент в итоговый массив

arrayms[i] = min;

lefts[min\_ind]++;

}

free(lefts);

free(rights);

free(data\_cpy);

return 0;

}

//главная функция

int main()

{

FILE\* input = fopen("input.txt", "r");

if (input == NULL)

{

printf("error: fopen\n");

return -1;

}

fscanf(input, "%d%d", &T, &N);

arrayms = (int\*)malloc(N \* sizeof(int));

if (arrayms == NULL)

{

printf("error: malloc\n");

return -1;

}

memset(arrayms, 0, N \* sizeof(int));

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

fscanf(input, "%d", &arrayms[i]);

fclose(input);

pthread\_t\* tids = (pthread\_t\*)malloc(T \* sizeof(pthread\_t));

if (tids == NULL)

{

printf("error: malloc\n");

free(arrayms);

return -1;

}

memset(tids, 0, T \* sizeof(pthread\_t));

input\_data = (param\_t\*)malloc(T \* sizeof(param\_t));

if (input\_data == NULL)

{

printf("error: malloc\n");

free(arrayms);

free(tids);

return -1;

}

memset(input\_data, 0, T \* sizeof(param\_t));

//расставление границ задачам

if (N < 1000)//если мало элементов, то считать будет один поток

{

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

{

input\_data[i].flag = -1;//не доступно для обработки

input\_data[i].l = -1;

input\_data[i].r = -1;

}

input\_data[0].flag = 1;

input\_data[0].l = 0;

input\_data[0].r = N - 1;

}

else

{

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

{

input\_data[i].l = N / T \* i + 1;

input\_data[i].r = N / T \* (i + 1);

input\_data[i].flag = 1;//доступно для обработки

}

input\_data[T - 1].r = N - 1;

input\_data[0].l = 0;

}

int i = 0;

for (; i < T; i++)

{

if (0 != pthread\_create(&tids[i], 0, thread\_entry, 0))

{

printf("pthread\_create failed. errno: %d\n", errno);

free(arrayms);

free(tids);

free(input\_data);

return -1;

}

}

struct timespec started, finished;

clock\_gettime(CLOCK\_REALTIME, &started);

for (i = 0; i < T; i++)

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

merge();

clock\_gettime(CLOCK\_REALTIME, &finished);

int time = to\_ms(&finished) - to\_ms(&started);

FILE\* output;

output = fopen("output.txt", "w");

if (output == NULL)

{

printf("error: fopen\n");

return -1;

}

fprintf(output, "%d\n%d\n", T, N);

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

if (i != N - 1) fprintf(output, "%d ", arrayms[i]);

else fprintf(output, "%d", arrayms[i]);

fclose(output);

output = fopen("time.txt", "w");

if (output == NULL)

{

printf("error: fopen\n");

return -1;

}

fprintf(output, "%d", time);

fclose(output);

free(tids);

free(input\_data);

free(arrayms);

return 0;

}

**ПРИЛОЖЕНИЕ Г**

**Листинг программы phil.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <cstring>

#include <cstdint>

#include <cstdlib>

#include <vector>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#define PHIL\_CNT 5

using namespace std;

unsigned long long get\_time(void);

unsigned long long get\_time(void)

{

struct timespec time\_interval;

clock\_gettime(CLOCK\_REALTIME, &time\_interval);

unsigned long long cur\_time = time\_interval.tv\_nsec / 1000000;

cur\_time += time\_interval.tv\_sec \* 1000;

return cur\_time;

}

unsigned long long started;

unsigned long long finished = 0;

unsigned long long sleep\_time = 0;

vector<int> call\_stack; // Стек вызовов номеров филосософов

const int phil\_num[PHIL\_CNT] = { 0, 1, 2, 3, 4 }; // Массив номеров философов

pthread\_t phil\_thread[PHIL\_CNT]; // Массив потоков философов

sem\_t phil\_sem[PHIL\_CNT]; // Массив семафоров философов

pthread\_t servant\_thread; // Поток слуги

sem\_t servant\_sem; // Семафор слуги (на 2 потока)

void threads\_init(int, char\*\*); // Инициализация дополнительных потоков, заполнения стека вызовов

void threads\_deinit(void); // Деинициализация потоков, освобождение памяти

void\* servant\_thread\_func(void\*); // Функция потока слуги

void\* phil\_thread\_func(void\*); // Функция потока философа

void threads\_init(int argc, char\*\* argv)

{

char\* enf\_of\_str = NULL;

//считается строка до символа окончания строки и переведется в систему счисления 10\

к результату добавится начальное время

finished = strtoull(argv[1], &enf\_of\_str, 10) + started;

sleep\_time = strtoull(argv[2], &enf\_of\_str, 10);

int change\_cnt = (finished - started) / sleep\_time \* 2;

int ind = 0;

for (int i = 0; ind < change\_cnt; i = (i + 2) % PHIL\_CNT, ind++)

call\_stack.push\_back(i);

for (int i = 0; i < PHIL\_CNT; i++)

sem\_init(&phil\_sem[i], 1, 0);

memset(&servant\_thread, 0, sizeof(pthread\_t));

memset(phil\_thread, 0, PHIL\_CNT \* sizeof(pthread\_t));

pthread\_create(&servant\_thread, 0, servant\_thread\_func, 0);

for (int i = 0; i < PHIL\_CNT; i++)

pthread\_create(phil\_thread + i, 0, phil\_thread\_func, (void\*)(phil\_num + i));

for (int i = 0; i < PHIL\_CNT; i++)

pthread\_join(phil\_thread[i], 0);

pthread\_join(servant\_thread, 0);

}

void threads\_deinit(void)

{

for (int i = 0; i < PHIL\_CNT; i++)

sem\_destroy(&phil\_sem[i]);

sem\_destroy(&servant\_sem);

}

void\* servant\_thread\_func(void\* param)

{

while (get\_time() < finished)

{

int phil\_1 = call\_stack.back();

call\_stack.pop\_back();

int phil\_2 = call\_stack.back();

call\_stack.pop\_back();

sem\_post(&phil\_sem[phil\_1]);

sem\_post(&phil\_sem[phil\_2]);

usleep(sleep\_time \* 1000);

sem\_wait(&phil\_sem[phil\_1]);

sem\_wait(&phil\_sem[phil\_2]);

}

for (uint8\_t i = 0; i < PHIL\_CNT; i++)

sem\_post(&phil\_sem[i]);

return 0;

}

void\* phil\_thread\_func(void\* num)

{

int phil\_num = \*(int\*)num;

while (true)

{

int res;

sem\_wait(&phil\_sem[phil\_num]);

if (get\_time()>=finished)

{

sem\_post(&phil\_sem[phil\_num]);

return 0;

}

printf("%llu:%u:T->E\n", get\_time() - started, phil\_num + 1);

usleep(sleep\_time \* 1000);

if (get\_time() >= finished)

{

printf("%llu:%u:E->T\n", get\_time() - started, phil\_num + 1);

sem\_post(&phil\_sem[phil\_num]);

return 0;

}

printf("%llu:%u:E->T\n", get\_time() - started, phil\_num + 1);

sem\_post(&phil\_sem[phil\_num]);

usleep(sleep\_time \* 1000);

}

return 0;

}

//TOTAL - общее время работы программы

//PHIL - минимальное время пребывания филосова в одном из состояний

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

{

started = get\_time();

threads\_init(argc, argv);

threads\_deinit();

return 0;

}