

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Операционные системы»

Управление потоками в ОС. Обеспечение синхронизации между
потоками

Студент: П. Ф. Гришин
Преподаватель: Е. С. Миронов
Группа: М8О-201Б-21
Вариант: 1
Дата:
Оценка:
Подпись:

Москва, 2023

1 Постановка задачи

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.

Отсортировать массив целых чисел при помощи битонической сортировки.

2 Сведения о программе

Программа написана на Си в Unix подобной операционной системе на базе ядра Linux. Для компиляции требуется ключ `-lpthread`, для запуска программы необходимо указать в качестве аргумента количество потоков, которые максимально могут быть использованы.

Программа сортирует массив с помощью битонной сортировки. Сначала пользователь должен ввести количество элементов массива. Далее должен передать все элементы.

Существует три глобальных переменных, которые являются мьютексом и 2 переменными количества потоков, используемые и максимальные. Их использование между потоками регулируется мьютексом. Сортируемый массив передается как указатель. И для него не нужен мьютекс, так как каждый поток работает на своем участке, не перекрываемом другими.

3 Общий метод и алгоритм решения

Параметром запуска программы мы указываем максимальное количество используемых потоков.

Далее мы указываем длину массива, и создаем массив долив длину до ближайшего числа 2^n . Так как это необходимо для работы алгоритма сортировки. Далее мы считываем данные, а оставшийся участок массива заполняем максимально возможными элементами, чтобы после сортировки они оказались в конце массива, и мы могли также легко их удалить.

Далее вызывается функция сортировки. Сначала мы рекурсивно разбиваем массив

по полам, задавая целевое направление сначала вниз, а потом вверх, чтобы получить битонную последовательность. При этом пока у нас есть свободные потоки мы выделяем под вторую половину отдельный поток, а когда они закончатся начинаем работать в однопотоке, на выделенном участке.

Когда мы достигаем массива из одного элемента его можно считать отсортированным, как по возрастанию так и по убыванию. Поэтому два соседних элемента можно считать битонной последовательностью, которую можно собрать в одну возрастающую или убывающую.

Начинаем объединять битонные последовательности. Находим расстояние между двумя элементами, которые будем сравнивать, оно равно половине участка массива. Далее мы проходим по половине участка и сравниваем каждый элемент с элементом через заданное расстояние. При необходимости меняем их местами. Далее разбиваем данный участок на 2 и повторяем слияние, и так пока его длина не станет равна 1.

Таким образом мы получаем битоническую последовательность большего размера, к которой можно опять применить слияние. Повторяем эту процедуру до окончания сортировки. При этом когда мы будем делать слияние для 2 участков, созданных разными потоками, слияние мы опять разбиваем на несколько потоков, пока это возможно.

При всем этом если один поток подготовил последовательность, а второй еще нет, то первый его ждет, из чего следует вывод, что увеличение производительности будет только при увеличении количества потоков до следующей степени двойки.

4 Листинг программы

main.c

```
1 | #include "library.h"
2 | #include "bitonic.h"
3 | #include "pthread.h"
4 | #include <sys/time.h>
5 |
6 | #define INVALID_INPUT 403
7 | #define WRONG_ANSWER 405
8 |
9 | #define MAXINT 2147483647
10 | #define UP 1
11 | #define DOWN 0
12 |
13 | pthread_mutex_t lock;
14 | size_t max_threads = 1;
15 | size_t use_threads = 1;
16 |
17 | void InitArgs(ArgsBitonic *args, int *array, int size, int start, int dir){
```

```

18     args->array = array;
19     args->size = size;
20     args->start = start;
21     args->dir = dir;
22 }
23
24
25 void Comparator(int *array, int i, int j, int dir){
26     if(dir == (array[i] > array[j])){
27         int temp = array[i];
28         array[i] = array[j];
29         array[j] = temp;
30     }
31 }
32
33 void BitonicMergeSinglThread(ArgsBitonic *args){
34     if(args->size > 1){
35         int nextsize = args->size / 2;
36         for(int i = args->start; i < nextsize + args->start; ++i){
37             Comparator(args->array, i, i + nextsize, args->dir);
38         }
39
40         ArgsBitonic args1;
41         ArgsBitonic args2;
42         InitArgs(&args1, args->array, nextsize, args->start, args->dir);
43         InitArgs(&args2, args->array, nextsize, args->start + nextsize, args->dir);
44
45         BitonicMergeSinglThread(&args1);
46         BitonicMergeSinglThread(&args2);
47     }
48 }
49
50 void BitonicSortSinglThread(ArgsBitonic *args){
51     if(args->size > 1){
52         int nextsize = args->size / 2;
53
54         ArgsBitonic args1;
55         ArgsBitonic args2;
56         InitArgs(&args1, args->array, nextsize, args->start, DOWN);
57         InitArgs(&args2, args->array, nextsize, args->start + nextsize, UP);
58
59         BitonicSortSinglThread(&args1);
60         BitonicSortSinglThread(&args2);
61         BitonicMergeSinglThread(args);
62     }
63 }
64
65 void BitonicMergeMultiThreads(ArgsBitonic *args){
66     if(args->size > 1){

```

```

67     int nextsize = args->size / 2;
68     int isParal = 0;
69     pthread_t tid;
70
71     for(int i = args->start; i < nextsize + args->start; ++i){
72         Comparator(args->array, i, i + nextsize, args->dir);
73     }
74
75     ArgsBitonic args1;
76     ArgsBitonic args2;
77     InitArgs(&args1, args->array, nextsize, args->start, args->dir);
78     InitArgs(&args2, args->array, nextsize, args->start + nextsize, args->dir);
79
80     pthread_mutex_lock(&lock);
81     if(use_threads < max_threads){
82         ++use_threads;
83         pthread_mutex_unlock(&lock);
84         isParal = 1;
85         pthread_create(&tid, NULL, (void*) &BitonicMergeMultiThreads, &args1);
86         BitonicMergeMultiThreads(&args2);
87     } else {
88         pthread_mutex_unlock(&lock);
89         BitonicMergeSinglThread(&args1);
90         BitonicMergeSinglThread(&args2);
91     }
92
93     if(isParal){
94         pthread_join(tid, NULL);
95         pthread_mutex_lock(&lock);
96         --use_threads;
97         pthread_mutex_unlock(&lock);
98     }
99 }
100 }
101
102 void BitonicSortMultiThreads(ArgsBitonic *args){
103     if(args->size > 1 ){
104         int nextsize = args->size / 2;
105         int isParal = 0;
106         pthread_t tid;
107
108         ArgsBitonic args1;
109         ArgsBitonic args2;
110         InitArgs(&args1, args->array, nextsize, args->start, DOWN);
111         InitArgs(&args2, args->array, nextsize, args->start + nextsize, UP);
112
113         pthread_mutex_lock(&lock);
114         if(use_threads < max_threads){
115             ++use_threads;

```

```

116         pthread_mutex_unlock(&lock);
117         isParal = 1;
118         pthread_create(&tid, NULL, (void*) &BitonicSortMultiThreads, &args1);
119         BitonicSortMultiThreads(&args2);
120     } else {
121         pthread_mutex_unlock(&lock);
122         BitonicSortSinglThread(&args1);
123         BitonicSortSinglThread(&args2);
124     }
125
126     if(isParal){
127         pthread_join(tid, NULL);
128         pthread_mutex_lock(&lock);
129         --use_threads;
130         pthread_mutex_unlock(&lock);
131     }
132     BitonicMergeMultiThreads(args);
133 }
134 }
135
136 void bitonicsort(int *array, int size, int threads){
137     pthread_mutex_init(&lock, NULL);
138
139     ArgsBitonic args;
140     InitArgs(&args, array, size, 0, UP);
141
142     if(threads > 1)
143         max_threads = threads;
144
145     BitonicSortMultiThreads(&args);
146
147     pthread_mutex_destroy(&lock);
148 }
149
150 int SizeStep(int Num){
151     int i = 1;
152     while(i < Num)
153         i *= 2;
154     return i;
155 }
156
157
158 int ThreadSort(int threads, char* nameF){
159     FILE* file;
160     if(threads < 1){
161         printf("ERROR: Incorrect number of threads\n");
162         return INVALID_INPUT;
163     }
164 }

```

```

165     int input_size;
166     file = fopen(nameF, "r");
167     if (file == NULL) printf("ERROR: Can't open this file\n");
168
169     fscanf(file, "%d", &input_size);
170     // 2^k >= input_size
171     int size_array = SizeStep(input_size);
172     int *array = malloc(sizeof(int)*size_array);
173
174     for(int i = 0; i < input_size; ++i)
175         fscanf(file, "%d", array+i);
176
177     fclose(file);
178
179     for(int i = input_size; i < size_array; ++i)
180         array[i] = MAXINT;
181
182
183     bitonicsort(array, size_array, threads);
184
185     for(int i = 1; i < input_size; ++i) {
186         if(array[i - 1] > array[i]) {
187             printf("ERROR: Something goes wrong. Array is not sorted\n");
188             return WRONG_ANSWER;
189         }
190     }
191     printf("Array is sorted\n");
192
193     free(array);
194     return 0;
195 }

```

5 Демонстрация работы программы

```

gpavel@gpavel-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/Desktop/OS/tests$ ./lab3_test
Running main() from /home/gpavel/Desktop/tmpo/tests/_deps/googletest-src/googletest/s
[=====] Running 3 tests from 2 test suites.
[-----] Global test environment set-up.
[-----] 2 tests from ThirdLabTests
[ RUN      ] ThirdLabTests.IncorrectNumberOfThreads
ERROR: Incorrect number of threads
ERROR: Incorrect number of threads
Passed the test for incorrect input of the number of threads
[      OK  ] ThirdLabTests.IncorrectNumberOfThreads (0 ms)
[ RUN      ] ThirdLabTests.SingleThread

```

```

Array is sorted
[      OK ] ThirdLabTests.SingleThread (1085 ms)
[-----] 2 tests from ThirdLabTests (1085 ms total)
[-----] 1 test from ThirdLabTest
[ RUN      ] ThirdLabTest.MultiThreads
Max thread count is 2
Array is sorted
Runtime is 604 ms

Max thread count is 3
Array is sorted
Runtime is 581 ms

Max thread count is 4
Array is sorted
Runtime is 567 ms

Max thread count is 5
Array is sorted
Runtime is 536 ms

7Max thread count is 6
Array is sorted
Runtime is 555 ms

Max thread count is 7
Array is sorted
Runtime is 551 ms

Max thread count is 8
Array is sorted
Runtime is 421 ms

Max thread count is 9
Array is sorted
Runtime is 362 ms
[      OK ] ThirdLabTest.MultiThreads (4182 ms)
[-----] 1 test from ThirdLabTest (4182 ms total)
[-----] Global test environment tear-down
[=====] 3 tests from 2 test suites ran. (5267 ms total)
[ PASSED ] 3 tests.

```



```
gpavel@gpavel-HP-Pavilion-Gaming-Laptop-17-cd1xxx:~/Desktop/OS/tests$
```

6 Вывод

Многие языки программирования позволяют пользователю работать с потоками. Создание потоков происходит быстрее, чем создание процессов, за счет того, что при создании потока не копируется область памяти, а они все работают с одной областью памяти. Поэтому многопоточность используют для ускорения не зависящих друг от друга, однотипных задач, которые будут работать параллельно.

Язык Си предоставляет данный функционал пользователям Unix-подобных операционных систем с помощью библиотеки `pthread.h`. Средствами языка Си можно совершать системные запросы на создание и ожидания завершения потока, а также использовать различные примитивы синхронизации.

В данной лабораторной работе был реализован и исследован алгоритм битонной сортировки. Установив при этом, что используя 9 потока можно молучить выигрыш по времени в 2 раза. При дальнейшем увеличении потоков прирост почти не увеличиваться, а даже может уменьшаться из-за того, что на управление и переключение потоков уходит больше времени, чем они выигрывают.