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 ОС. Обеспечение синхронизации между потоками.

Студент: С. А. Арапов
Преподаватель: Е. С Миронов
Группа: М8О-208Б-19

Дата: Оценка:
Подпись:

Москва, 2021

Лабораторная работа №3

Тема: Приобритение практических навыков в:

- Управлении потоками в ОС
- Обеспечение синхронизации потоков

Задача: Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки 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.

Вариант №1: Отсортировать массив целых чисел при помощи битонической сортировки.

1 Описание программы

Код программы написан на языке Си, в ОС на базе ядра Linux. Для сборки программы требуется ключ -lpthread. Во время запуска откомпилированной программы нужно указать количество потоков, максимальное количество которых может быть использовано.

Во время работы программы для начала указывается число элементов массива, а затем и сами элементы.

На вывод программа подаёт отсортированную с помощью битонической сортировки последовательность чисел.

2 Алгоритм решения

С запуском программы указывается максимально возможно количество потоков, используемых во время выполнения. Указывается длина массива. Если длина массива не является степенью двойки, то длина дополняется до этого числа, так как это необходимое условие для сортировки. Затем считываются данные, а оставшееся место заполняется максимальным элементом. Далее приступаем к самой сортировке данных.

Алгоритм основан на сортировке битоннических последовательностей. Битонической последовательностью называется последовательность, которая сначала монотонно не убывает, а затем монотонно не возрастает. Есть два этапа: формирование битонической последовательности и создание одной отсортированной последовательности из битонической последовательности. После первого шага первая половина сортируется в порядке возрастания, а вторая половина - в порядке убывания. Мы сравниваем первый элемент первой половины с первым элементом второй половины, затем второй элемент первой половины со вторым элементом второй и так далее. Мы меняем элементы, если элемент первой половины меньше. После вышеуказанных шагов сравнения и обмена мы получаем две битонные последовательности в массиве. Продолжаем так делать пока вся последовательность не будет отсортированна.

Массив рекурсивно разбивается пополам, задавание направление на возрастание и убывание соответственно для получения битонической последовательности. При этом пока есть свободные потоки они выделяются под вторую половину массива. Если же максимум потоков достигнут то просто работаем в одном потоке на выделенном участке. По достижении участка в один элемент считаем его отсортированным, поэтому два элемента можно превратить в бетонную последовательность. Далее объединиям наши последовательности. Сравниваем первый элемент массива и средний если необходимо менеяем их местами. И так далее. При этом для слияния аналогично выделяем потоки, пока они есть. Если один поток завершил свою работу и получил отсортированную последовательность, а второй еще нет, то первый дожидается выполнения второго. Отсюда можно сделать вывод, что производительность будет увеличиваться только когда очередное количество потоков будет равняться степени двойки.

3 Исходный код

main.c

```
1  #include "stdio.h"
2  #include "stdlib.h"
3  #include "bitonic.h"
4  #include <sys/time.h>
5
6  #define MAXINT 2147483647
7
8  #define timer
9
10 int SizeStep(int Num){
11     int i = 1;
12     while(i < Num)
13         i *= 2;
14     return i;
```

```
}


int main(int argc, char *argv[]){
    int threads = 1;

    if(argc == 2){
        threads = atoi(argv[1]);
    }

    int input_size;
    scanf("%d",&input_size);

    //   2^k >= input_size
    int size_array = SizeStep(input_size);
    int *array = malloc(sizeof(int)*size_array);

    for(int i = 0; i < input_size; ++i)
        scanf("%d",array+i);
    for(int i = input_size; i < size_array; ++i)
        array[i] = MAXINT;

    #ifdef timer
    struct timeval startwtime, endwtime;
    gettimeofday(&startwtime, NULL);
    #endif

    bitonicsort(array, size_array, threads);

    #ifdef timer
    gettimeofday(&endwtime, NULL);
    double time = (double)((endwtime.tv_usec - startwtime.tv_usec)/1.0e6 + endwtime.
        tv_sec - startwtime.tv_sec);
    printf("%f\n", time);
    #endif

    #ifndef timer
    for(int i=0;i<input_size;++i){
        printf("%d\n",array[i]);
    }
    #endif

    free(array);
    return 0;
}
```

bitonic.c

```
#include "pthread.h"
#include "bitonic.h"
```

```
3  #include "stdio.h"
4
5  #define UP 1
6  #define DOWN 0
7
8  pthread_mutex_t lock;
9  size_t max_threads = 1;
10 size_t use_threads = 1;
11
12 void InitArgs(ArgsBitonic *args, int *array, int size, int start, int dir){
13     args->array = array;
14     args->size = size;
15     args->start = start;
16     args->dir = dir;
17 }
18
19
20 void Comparator(int *array, int i, int j, int dir){
21     if(dir == (array[i] > array[j])){
22         int temp = array[i];
23         array[i] = array[j];
24         array[j] = temp;
25     }
26 }
27
28 void BitonicMergeSingleThread(ArgsBitonic *args){
29     if(args->size > 1){
30         int nextsize = args->size / 2;
31         for(int i = args->start; i < nextsize + args->start; ++i){
32                 Comparator(args->array, i, i + nextsize, args->dir);
33         }
34
35         ArgsBitonic args1;
36         ArgsBitonic args2;
37         InitArgs(&args1, args->array, nextsize, args->start, args->dir);
38         InitArgs(&args2, args->array, nextsize, args->start + nextsize, args->dir);
39
40         BitonicMergeSingleThread(&args1);
41         BitonicMergeSingleThread(&args2);
42     }
43 }
44
45 void BitonicSortSingleThread(ArgsBitonic *args){
46     if(args->size > 1){
47         int nextsize = args->size / 2;
48
49         ArgsBitonic args1;
50         ArgsBitonic args2;
51         InitArgs(&args1, args->array, nextsize, args->start, DOWN);
```

```
52          InitArgs(&args2, args->array, nextsize, args->start + nextsize, UP);
53
54          BitonicSortSingleThread(&args1);
55          BitonicSortSingleThread(&args2);
56          BitonicMergeSingleThread(args);
57      }
58  }
59
60  void BitonicMergeMultiThreads(ArgsBitonic *args){
61      if(args->size > 1){
62          int nextsize = args->size / 2;
63          int isParal = 0;
64          pthread_t tid;
65
66          for(int i = args->start; i < nextsize + args->start; ++i){
67                  Comparator(args->array, i, i + nextsize, args->dir);
68          }
69
70          ArgsBitonic args1;
71          ArgsBitonic args2;
72          InitArgs(&args1, args->array, nextsize, args->start, args->dir);
73          InitArgs(&args2, args->array, nextsize, args->start + nextsize, args->dir);
74
75          pthread_mutex_lock(&lock);
76          if(use_threads < max_threads){
77              ++use_threads;
78              pthread_mutex_unlock(&lock);
79              isParal = 1;
80              pthread_create(&tid, NULL,(void*) &BitonicMergeMultiThreads, &args1);
81              BitonicMergeMultiThreads(&args2);
82          } else {
83              pthread_mutex_unlock(&lock);
84              BitonicMergeSingleThread(&args1);
85              BitonicMergeSingleThread(&args2);
86          }
87
88          if(isParal){
89              pthread_join(tid, NULL);
90              pthread_mutex_lock(&lock);
91              --use_threads;
92              pthread_mutex_unlock(&lock);
93          }
94      }
95  }
96
97  void BitonicSortMultiThreads(ArgsBitonic *args){
98      if(args->size > 1 ){
99          int nextsize = args->size / 2;
100         int isParal = 0;
```

```
        pthread_t tid;

        ArgsBitonic args1;
        ArgsBitonic args2;
        InitArgs(&args1, args->array, nextsize, args->start, DOWN);
        InitArgs(&args2, args->array, nextsize, args->start + nextsize, UP);

        pthread_mutex_lock(&lock);
        if(use_threads < max_threads){
            ++use_threads;
            pthread_mutex_unlock(&lock);
            isParal = 1;
            pthread_create(&tid, NULL,(void*) &BitonicSortMultiThreads, &args1);
            BitonicSortMultiThreads(&args2);
        } else {
            pthread_mutex_unlock(&lock);
            BitonicSortSingleThread(&args1);
            BitonicSortSingleThread(&args2);
        }

        if(isParal){
            pthread_join(tid, NULL);
            pthread_mutex_lock(&lock);
            --use_threads;
            pthread_mutex_unlock(&lock);
        }
        BitonicMergeMultiThreads(args);
    }
}

void bitonicsort(int *array, int size, int threads){
    pthread_mutex_init(&lock, NULL);

    ArgsBitonic args;
    InitArgs(&args,array,size,0,UP);

    if(threads > 1)
        max_threads = threads;

    BitonicSortMultiThreads(&args);

    pthread_mutex_destroy(&lock);
}
```

bitonic.h

```
#pragma once

#include "pthread.h"

```

```c
#define UP 1
#define DOWN 0

typedef struct ArgsBitonic{
    int *array;
    int size;
    int start;
    int dir;
}ArgsBitonic;

void InitArgs(ArgsBitonic *args, int *array, int size, int start, int dir);
void Comparator(int *array, int i, int j, int dir);
void BitonicMergeSingleThread(ArgsBitonic *args);
void BitonicSortSingleThread(ArgsBitonic *args);
void BitonicMergeMultiThreads(ArgsBitonic *args);
void BitonicSortMultiThreads(ArgsBitonic *args);
void bitonicsort(int *array, int size, int threads);
```

makefile

```
CC = gcc
CFLAGS = -c -Wall
FINALFLAGS = -pthread -o

all: main

main: main.o bitonic.o
  $(CC) main.o bitonic.o $(FINALFLAGS) main
main.o: main.c bitonic.h
  $(CC) $(CFLAGS) main.c
bitonic.o: bitonic.c bitonic.h
  $(CC) $(CFLAGS) bitonic.c
clean:
  rm -r *.o main
```

4 Запуск программы и демонстрация работы

```
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ make
gcc -c -Wall main.c
gcc -c -Wall bitonic.c
gcc main.o bitonic.o -pthread -o main
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ls
ans  bitonic.c  bitonic.h  bitonic.o  main  main.c  main.o  makefile  test
test_generator.py
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 4
10
```

```
8 7 1 2 4 -1 10 3 9 -8
-8
-1
1
2
3
4
7
8
9
10
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ strace -f -e trace="%process,write"
-o log ./main 4
10
8 7 1 2 4 -1 10 3 9 -8
-8
-1
1
2
3
4
7
8
9
10
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ cat log

1616  execve("./main",["./main","4"],0x7fffd37b4380 /* 19 vars */) = 0
1616  arch_prctl(0x3001 /* ARCH_??? */,0x7fffec90bb40) = -1 EINVAL (Invalid
argument)
1616  arch_prctl(ARCH_SET_FS,0x7f014fc70740) = 0
1616  clone(child_stack=0x7f014fc5ffb0,flags=CLONE_VM|
CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|
CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
parent_tid=[1617],tls=0x7f014fc60700,child_tidptr=0x7f014fc609d0) = 1617

1616  clone(child_stack=0x7f014f44ffb0,flags=CLONE_VM|CLONE_FS|
CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,parent_tid=[1618],
tls=0x7f014f450700,child_tidptr=0x7f014f4509d0) = 1618
```

```
1618  exit(0)                           = ?
1617  clone(child_stack=0x7f014ec3ffb0,flags=CLONE_VM|CLONE_FS|
CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID <unfinished ...>

1618  +++ exited with 0 +++
1616  clone(child_stack=0x7f014f44ffb0,flags=CLONE_VM|CLONE_FS|
CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID <unfinished ...>

1617  <... clone resumed>,parent_tid=[1619],
tls=0x7f014ec40700,child_tidptr=0x7f014ec409d0) = 1619

1616  <... clone resumed>,parent_tid=[1620],
tls=0x7f014f450700,child_tidptr=0x7f014f4509d0) = 1620

1619  exit(0)                           = ?
1620  exit(0 <unfinished ...>
1619  +++ exited with 0 +++
1620  <... exit resumed>)               = ?
1617  clone(child_stack=0x7f014ec3ffb0,flags=CLONE_VM|CLONE_FS|
CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID <unfinished ...>

1620  +++ exited with 0 +++
1617  <... clone resumed>,parent_tid=[1621],
tls=0x7f014ec40700,child_tidptr=0x7f014ec409d0) = 1621

1617  clone(child_stack=0x7f014f44ffb0,flags=CLONE_VM|CLONE_FS
|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,parent_tid=[1622],
tls=0x7f014f450700,child_tidptr=0x7f014f4509d0) = 1622

1621  exit(0)                           = ?
1621  +++ exited with 0 +++
1622  exit(0)                           = ?
1622  +++ exited with 0 +++
1617  exit(0)                           = ?
1617  +++ exited with 0 +++
1616  clone(child_stack=0x7f014fc5ffb0,flags=CLONE_VM|CLONE_FS|
CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
```

```
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,parent_tid=[1623],
tls=0x7f014fc60700,child_tidptr=0x7f014fc609d0) = 1623

1616  clone(child_stack=0x7f014ec3ffb0,flags=CLONE_VM|CLONE_FS|
CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID <unfinished ...>

1623  clone(child_stack=0x7f014f44ffb0,flags=CLONE_VM|CLONE_FS|
CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|
CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID <unfinished ...>

1616  <... clone resumed>,parent_tid=[1624],
tls=0x7f014ec40700,child_tidptr=0x7f014ec409d0) = 1624

1623  <... clone resumed>,parent_tid=[1625],
tls=0x7f014f450700,child_tidptr=0x7f014f4509d0) = 1625

1624  exit(0)                           = ?
1625  exit(0 <unfinished ...>
1624  +++ exited with 0 +++
1625  <... exit resumed>)               = ?
1625  +++ exited with 0 +++
1623  exit(0)                           = ?
1623  +++ exited with 0 +++
1616  write(1,"-8",3)               = 3
1616  write(1,"-1",3)               = 3
1616  write(1,"1",2)                = 2
1616  write(1,"2",2)                = 2
1616  write(1,"3",2)                = 2
1616  write(1,"4",2)                = 2
1616  write(1,"7",2)                = 2
1616  write(1,"8",2)                = 2
1616  write(1,"9",2)                = 2
1616  write(1,"10",3)               = 3
1616  exit_group(0)                     = ?
1616  +++ exited with 0 +++
```

5 Тест производительности

Проверять будем на тестах размером в 500000, 2500000, 5000000 элементов и на 8-ми ядерном процессоре.

```
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ nproc
8

ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ python3 test_generator.py
500_000
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 1 <test
0.708356
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 2 <test
0.426568
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 3 <test
0.441195
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 4 <test
0.270915
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 5 <test
0.265912
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 6 <test
0.245091
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 7 <test
0.265175
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 8 <test
0.261323

ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ python3 test_generator.py
2_500_000
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 1 <test
15.512694
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 2 <test
5.725034
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 3 <test
6.461111
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 4 <test
2.995543
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 5 <test
2.711381
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 6 <test
2.581098
```

```
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 7 <test
2.619003
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 8 <test
2.194820

ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ python3 test_generator.py
5_000_000
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 1 <test
31.580048
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 2 <test
14.965795
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 3 <test
15.951123
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 4 <test
7.784170
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 5 <test
7.850845
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 6 <test
6.987806
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 7 <test
7.637479
ogrocket@LAPTOP-ADULJM7A:/mnt/d/OS/OS/os_lab3$ ./main 8 <test
8.150856
```

Таблица 1. Исследование на 500000 элементах.

Количество потоков(n)	Время работы	Ускорение ($S_n = T_1/T_n$)	Эффективность ($X_n = S_n/n$)
1	0.708356	-	-
2	0.426568	1.66	0.83
3	0.441195	1.61	0.54
4	0.270915	2.61	0.65
5	0.265912	2.66	0.53
6	0.245091	2.89	0.48
7	0.265175	2.67	0.38
8	0.265175	2.67	0.33

Таблица 2. Исследование на 1000000 элементах.

Количество потоков(n)	Время работы	Ускорение ($S_n = T_1/T_n$)	Эффективность ($X_n = S_n/n$)
1	15.512694	-	-
2	5.725034	2.71	1.35
3	6.461111	2.4	0.8
4	2.995543	5.18	1.29
5	2.711381	5.72	1.14
6	2.581098	6.01	1.0
7	2.619003	5.92	0.85
8	2.19482	7.07	0.88

Таблица 3. Исследование на 10000000 элементах.

Количество потоков(n)	Время работы	Ускорение ($S_n = T_1/T_n$)	Эффективность ($X_n = S_n/n$)
1	31.580048	-	-
2	14.965795	2.11	1.06
3	15.951123	1.98	0.66
4	7.78417	4.06	1.01
5	7.850845	4.02	0.8
6	6.987806	4.52	0.75
7	7.637479	4.13	0.59
8	8.150856	3.87	0.48

6 Выводы

Существует множество языков, позволяющее работать с потоками. Потоки создаются быстрее чем процессы, по той причине что потоки работают с одной областью памяти и она не копируется. Именно поэтому распараллеливание однотипных и не зависящих друг от друга задач так эффективно (яркий пример перемножение матриц в библиотеке numpy для языка python3). В языке Си работа с потоками осуществленна в библиотеке pthread.h, в которой можно создавать и завершать потоки, синхронизировать их. В бенчмарке я показал, что использование нескольких потоков может ускорить время работы 2-7 раз. При всё большем увеличении числа потоков прирост эффективности начинает снижаться, что еще раз подтверждает закон Амдала-Уэра об ограничении роста производительности.