Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Операционные системы»

Студент: С. М. Бокоч
Преподаватель: А. А. Кухтичев
Группа: М8О-204Б

Дата:
Оценка:
Подпись:

Москва, 2017

Лабораторная работа №3

Задача: Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). При создании необходимо предусмотреть ключи, которые позволяли бы задать максимальное количество потоков, используемое программой. При возможности необходимо использовать максимальное количество возможных потоков. Ограничение потоков может быть задано или ключом запуска вашей программы, или алгоритмом.

Вариант задания 4: Отсортировать массив строк при помощи TimSort.

1 Метод решения

Timsort — гибридный алгоритм сортировки, сочетающий сортировку вставками и сортировку слиянием,

1 Алгоритм сортировки

1. По специальному алгоритму входной массив разделяется на подмассивы.
2. Каждый подмассив сортируется **сортировкой вставками**.
3. Отсортированные подмассивы собираются в единый массив с помощью **сортировки слиянием**.

Перед сортировкой нам необходимо вычислить оптимальный размер подмассивов, с помощью функции **GetRun(n)**, *где* **n** - *размер исходного массива.* Если размер меньше 64 элементов, то получится обычная сортировка вставкой. На общем методе сортировке закончим.

2 Применение потоков

Мы можем применить потоки в сорировке два раза.
Я использую для потоков(нитей) следующие вызовы:

int pthread_create(pthread_t *thr, const pth_attr_t *attr, void* (*start)(void*), void *arg)
Создание потока. Первый аргумент этой функции thr - это указатель на переменную, в которую будет записан идентификатор созданного потока, который в последствии можно будет передавать другим вызовам, когда мы захотим сделать что-либо с этим потоком. Второй аргумент этой функции attr – это указатель на переменную, которая задает набор некоторых свойств создаваемого потока. Третий аргумент вызова это указатель на функцию типа void* ()(void *). Именно эту функцию и начинает выполнять вновь созданный поток, при этом в качестве параметра этой функции передается четвертый аргумент вызова. Функция pthread_create возвращает нулевое значение в случае успеха и ненулевой код ошибки в случае неудачи.
int pthread_join(pthread_t thread, void** value_ptr)
Эта функция дожидается завершения нити с идентификатором thread, и записывает ее возвращаемое значение в переменную на которую указывает value_ptr. При этом освобождаются все ресурсы связанные с потоком, и следовательно эта функция может быть вызвана для даннного потока только один раз.

Применение потоков поисходит в следующих случаях:

- Первый раз создаем потоки при сортировке подмассивов вставкой, передавая в **pthread_create** i поток, ссылку на функцию сортировкой вставкой и элемент, который преставляет собой структуру из левой и правой границы массива (массив объявлен глобально). Обязательно ждем завершения потоков.
- Второй раз, постепенно сливая все подмассивы.

Примечание: ввиду того, что количество используемых потоков ограничено, заводим дополнительную переенную *shift* задающее смещение, т.е. если у нас все потоки находятся в использовании, то ожидаем первый запущенный поток и т.д. (в первом и во втором случае).

Так как применение потоков простое до безобразия, то ошибок синхронизации быть не может. Так бы пришлось задействовать мьютексы и моя реализация стала бы сложнее для восприятия.

2 Листинг кода

TimSort.cpp

```
1  // C++ program to perform TimSort.
2  #include <cstdio>
3  #include <iostream>
4  #include <stdlib.h>
5  #include <pthread.h>
6  using namespace std;
7  int sizeThreads;
8  int RUN;
9  int n;
10 inline int GetRun(int n);
11 struct Range {
12   int left;
13   int right;
14 };
15 struct Merge2Arr {
16   int left;
17   int mid;
18   int right;
19 };
20 int *arr;
21 // this function sorts array from left index to
22 // to right index which is of size atmost RUN
23 void *insertionSort(void *arg);
24 // merge function merges the sorted runs
25 void *merge(void *arg);
26 // iterative Timsort function to sort the
27 // array[0...n-1] (similar to merge sort)
28 void TimSort();
29 // utility function to print the Array
30 void printArray(int arr[], int n);
31 // Driver program to test above function
32 int main(int argc, char **argv)
33 {
34   if (argc != 2) {
35     printf("Usage: ThreadsNumber\n");
36     exit(EXIT_FAILURE);
37   }
38   sizeThreads = atoi(argv[1]);
39   if (sizeThreads <= 0) {
40     printf("ERROR: usage ThreadNumber > 0\n");
41     exit(EXIT_FAILURE);
42   }
43   std::cin >> n;
44   arr = (int *)malloc(sizeof(int)*n);
45   for (int i = 0; i < n; i++) {
46         std::cin >> arr[i];
```

```
47      }
48    timSort();
49    printArray(arr, n);
50    free(arr);
51      return 0;
52  }
53  void TimSort()
54  {
55    RUN = GetRun(n);
56    // ====================> Sort individual subarrays of size RUN <====================
57    pthread_t *thread = (pthread_t *)malloc(sizeThreads*sizeof(pthread_t *));
58    printf("Thread Count %d\n", sizeThreads);
59    Range k[sizeThreads];
60    int count = 0;
61    int shifted = 0;
62      for (int i = 0; i < n; i+=RUN) {
63      if (count == sizeThreads) {
64        count = 0;
65        shifted++;
66      }
67      if (shifted > 0) {
68        pthread_join(thread[count], NULL);
69      }
70      k[count].left = i;
71      k[count].right = min(i + RUN - 1, n - 1);
72      pthread_create(&thread[count], NULL, insertionSort, &k[count]);
73      count++;
74    }
75    //==================> Wait all thread <======================
76    if (shifted > 0) {
77      count = sizeThreads;
78    }
79    for (int i = 0; i < count; i++) {
80      pthread_join(thread[i], NULL);
81    }
82    //==================> Merging all subarray <====================
83    Merge2Arr Index[sizeThreads/2];
84    for (int size = RUN; size < n; size = 2*size)
85      {
86      shifted = 0;
87      count = 0;
88      for (int left = 0; left < n; left += 2*size)
89          {
90        if (count == sizeThreads) {
91          count = 0;
92          shifted++;
93        }
94        if (shifted > 0) {
95          pthread_join(thread[count], NULL);
```

```
96        }
97
98        Index[count].right = min((left + 2*size - 1), (n-1));
99        Index[count].left = left;
100       Index[count].mid = (Index[count].left+Index[count].right)/2 ;
101       pthread_create(&thread[count], NULL, merge, &Index[count]);
102       count++;
103     }
104
105     if (shifted > 0) {
106       count = sizeThreads;
107     }
108     for (int i = 0; i < count; i++) {
109       pthread_join(thread[i], NULL);
110     }
111   }
112   free(thread);
113 }
114 void *insertionSort(void *arg)
115 {
116   Range *tmp = (Range *) arg;
117   int left = tmp->left;
118   int right = tmp->right;
119     for (int i = left + 1; i <= right; i++)
120     {
121         int temp = arr[i];
122         int j = i - 1;
123         while (arr[j] > temp && j >= left)
124         {
125             arr[j+1] = arr[j];
126             j--;
127         }
128         arr[j+1] = temp;
129   }
130 }
131 void *merge(void *arg)
132 {
133     // original array is broken in two parts
134   // left and right array
135   Merge2Arr *tmp = (Merge2Arr *) arg;
136     int len1 = tmp->mid - tmp->left + 1, len2 = tmp->right - tmp->mid;
137     int left[len1], right[len2];
138     for (int i = 0; i < len1; i++)
139         left[i] = arr[tmp->left + i];
140     for (int i = 0; i < len2; i++)
141         right[i] = arr[tmp->mid + 1 + i];
142     int i = 0;
143     int j = 0;
144     int k = tmp->left;
```

```
    // after comparing, we merge those two array
    // in larger sub array
    while (i < len1 && j < len2)
    {
        if (left[i] <= right[j])
        {
            arr[k] = left[i];
            i++;
        }
        else
        {
            arr[k] = right[j];
            j++;
        }
        k++;
    }
    // copy remaining elements of left, if any
    while (i < len1)
    {
        arr[k] = left[i];
        k++;
        i++;
    }
    // copy remaining element of right, if any
    while (j < len2)
    {
        arr[k] = right[j];
        k++;
        j++;
  }
}

inline int GetRun(int n) {
    int r = 0;
    while (n >= 64) {
        n >>= 1;
        r |= n & 1;
    }
    return n + r;
}


void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

3 Тест работоспособности

Я проверял на 10^6 тестах с помощью небольшого скрипта на python3. Приведу небольшой пример на 20 элементах.

```
[bokoch@MacKenlly codeforces]$ g++ -pthread TimSort.cpp -o timSort
[bokoch@MacKenlly codeforces]$ ./timSort
Usage: ThreadsNumber
[bokoch@MacKenlly codeforces]$ ./timSort 10
20
7 4 1 -8 9 6 4 1 2 3
8 14 25 2130 451 -455 12 333 555 88
Thread Count 10
-455  -8  1  1  2  3  4  4  6  7  8  9  12  14  25  88  333  451  555  2130
[bokoch@MacKenlly codeforces]$ ./timSort 0
ERROR: usage ThreadNumber > 0
```

4 Тест производительности

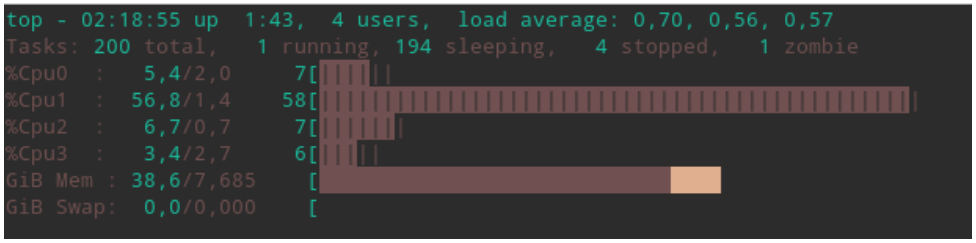
Тестирование производилось путем сравнения реализации TimSort с помощью потоков и реализации без потоков(из чистого интереса). Результаты меня удивили. Тестирование производилось на машине с CPU: Intel Core i7-4500U @ 4x 3GHz, RAM: 7869MiB.
Количество используемых потоков – 10.

Размер массива	SortWithThread	BasedTimSort
5000	0.003913	0.000461
25000	0.01598	0.003103
625000	0.072015	0.017341
3125000	0.481235	0.100467

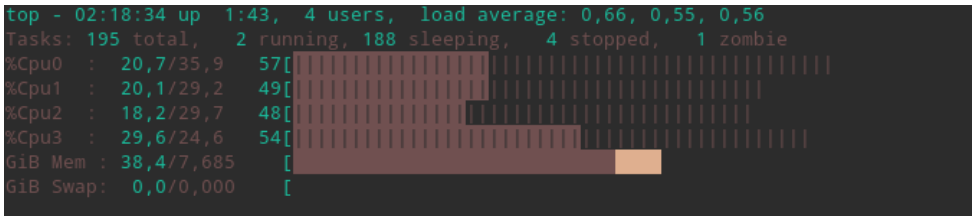
По таблице видно, что с потоками сортировка работает медленней, даже очень. Почитав об этом, я узнал, что время на создание нового потока требует времени и дополнительны ресурсов на его содержание.
Так как по алгоритму сортировки **всегда подмассивы содержат не больше 64 элемента**, из логики понятно, что потоков требуется очень много. Если бы подмассивы были большего размера, то, вероятно, реализация с потоками работала бы быстрее.

Ниже приведены картинки загрузки работы ядер процессора при:

1. Одном потоке



2. Многих потоках



5 Выводы

Безусловно, многопоточность – это великолепная идея. Достаточно взглянуть на большинство серверных приложений: в них потоки позволяют изолировать одинаковые участки программы для разных данных. К примеру, очередь на прием к врачу в больнице: если на месте один врач(поток), то клиенты(данные) очень долго будут ждать своей очереди, нежели если врачей было бы несколько.
Одним из главных недостатков использования потоков является сложность отладки программы, и синхронизации потоков между собой (чтобы они не обращались к одним и тем же данным в один и тот же момент времени).
Из достоинств можно отметить ускорение работы приложений, использующих ввод/обработку/вывод данных за счет возможости распределения этих операций по отдельным потокам. Это дает возвожность не прекращать выполнение программы во время возможных простоев из-за ожидания при чтении/записи данных.
Было полезно и информативно использовать POSIX thread, набил руку и немного наловчился писать не слишком замысловатые программы с использованием потоков.
Пару слов скажу о сортировке. Она работает быстрей QuickSort на почти упорядоченном массиве. Да и сама «гибридный» алгоритм довольно странный, не смотря на что она используется по стандарту в Python.