Василенко Анатолий
421 группа

Тестирование под Linux:

1. Характеристики системы:
    1. Процессор - Core 2 Duo - 2 ядра по 2,4 GHz (используется конечно же одно (за исключением того, что некоторые вызовы оси могут быть паралельны))
    2. Linux x64 - установлен на вирутальную машину из под vmware
    3. Процессор поддерживает Intel VT (Virutalization Technology)
2. Замеры при помощи утилиты time:
    1. Было сделано 10 замеров:

real	0m1.028s	0m1.089s	0m1.048s	0m1.092s	0m0.942s	0m1.047s	0m1.139s	0m0.986s	0m1.078s	0m0.992s
user	0m0.664s	0m0.612s	0m0.608s	0m0.608s	0m0.588s	0m0.604s	0m0.580s	0m0.508s	0m0.620s	0m0.612s
sys	0m0.360s	0m0.472s	0m0.432s	0m0.484s	0m0.348s	0m0.444s	0m0.552s	0m0.468s	0m0.452s	0m0.380s

    2. Среднее значение:
        1. real =1,0441s
        2. user=0,6004s
        3. sys=0,4392s
    3. После изменения порядка прохода по массиву (a[i][k] -> a[k][i]):

real	0m1.409s	0m1.366s	0m1.347s	0m1.360s	0m1.335s	0m1.397s	0m1.334s	0m1.335s	0m1.409s	0m1.329s
user	0m1.140s	0m1.092s	0m1.088s	0m1.104s	0m1.048s	0m1.160s	0m1.112s	0m1.084s	0m1.104s	0m1.076s
sys	0m0.264s	0m0.268s	0m0.256s	0m0.252s	0m0.284s	0m0.232s	0m0.220s	0m0.244s	0m0.296s	0m0.252s

    4. Среднее значение:
        1. real=1,3621s
        2. user=1,1008s
        3. sys=0,2568s

    Объяснение замеров:
    1. время real не всегда совпадает в временем user + sys, это может быть связано с тем, что real – это время которое прошло, user – это процессорное время которое было затрачено на работу программы пользователя, а sys время, затраченное на работу системы (вернее системных вызовов)

       Таким образом, если система многопроцессорная, то время real может быть меньше, чем время user+sys, а может быть и больше, если переключение контекста происходило слишком часто.
    2. Время работы после изменения порядка прохода по массиву (я рассматриваю время user, потому что время sys слишком сильно зависит от того, что происходит в операционной системе, а это неподконтрольные процессы) изменилось в среднем примерно на 0,5004s. Это связанно с тем, что размер массива, с которым работает программа, равен 381 Мб (4*10000*10000 (4 байта – размер float)), в то время, как L1 кеш используемого процессора равен 128 Кб, а L2 = 3Мб, что приводит к многократной перезаписи кеша, и увеличивает время выполнения user.

       Количество используемой памяти было выявлено и практически через скрипт:

       ```
       #! /bin/bash
       ./b &
       pmap -d $!
       ```

       Который вывел объём аллоцированной памяти процессом в 394564K
3. Замеры при помощи утилиты gprof

   Данная утилита может выводить количество времени, которое выполнялась та или иная функция в программе и общее количество времени, которое работала программа (конечно же есть ещё

огромное количество флагов отвечающих за разные опции, особенности и модификации профилирования, но основная задача именно такая)

Для использования этой утилиты, необходимо скомпилировать программу с флагом gcc –pg

Теперь после обычного запуска программы будет сгенерирован файл под названием gmon.out. И после этого можно будет проанализировать произошедшее при помощи gprof.

Сначала, я укажу основную расшифровку значений, которые выдаёт gprof, после чего укажу сами значения.

1. Расшифровка:
    1. **% time** - the percentage of the total running time of the program used by this function.
    2. **Cumulative seconds** - a running sum of the number of seconds accounted for by this function and those listed above it.
    3. **Self seconds** - the number of seconds accounted for by this function alone. This is the major sort for this listing.
    4. **Calls** - the number of times this function was invoked, if this function is profiled, else blank.
    5. **Self ms/call** - the average number of milliseconds spent in this function per call, if this function is profiled, else blank.
    6. **Total ms/call** - the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.
    7. **Name** - the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.
2. gprof -z выдаёт список всех функций в программе и время их работы:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
101.20      0.67     0.67                             main
  0.00      0.67     0.00                             __do_global_dtors_aux
  0.00      0.67     0.00                             __do_global_dtors_aux_fini_array_entry
  0.00      0.67     0.00                             __frame_dummy_init_array_entry
  0.00      0.67     0.00                             __gmon_start__
  0.00      0.67     0.00                             __libc_csu_fini
  0.00      0.67     0.00                             __libc_csu_init
  0.00      0.67     0.00                             _fini
  0.00      0.67     0.00                             _init
  0.00      0.67     0.00                             _start
  0.00      0.67     0.00                             atexit
  0.00      0.67     0.00                             call_gmon_start
  0.00      0.67     0.00                             data_start
  0.00      0.67     0.00                             deregister_tm_clones
  0.00      0.67     0.00                             frame_dummy
  0.00      0.67     0.00                             register_tm_clones
```

   Из таблицы видно, что основное время выполнения занимает функция main, которая перебирает все элементы массива. В то время, как служебные функции обёртки почти не занимают времени.
3. Можно было вывести информацию и только по фунции main, используя вызов gproof –f main :

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
101.20      0.67     0.67                             main
```

4. Можно так же просто ограничиться списком функций в программе используя gprof –r :

```
_init
_start
```

__gmon_start__
call_gmon_start
deregister_tm_clones
register_tm_clones
__do_global_dtors_aux
frame_dummy
main
__libc_csu_fini
__libc_csu_init
atexit
_fini
__frame_dummy_init_array_entry
__do_global_dtors_aux_fini_array_entry
data_start

5. Если изменить порядок индексации при переборе массива (a[i][k] -> a[k][i]):

```
1.  Each sample counts as 0.01 seconds.
2.   %  cumulative  self              self     total
3.  time  seconds  seconds   calls  Ts/call  Ts/call  name
4.  101.20    1.40    1.40                             main
5.    0.00    1.40    0.00                             __do_global_dtors_aux
6.    0.00    1.40    0.00                             __do_global_dtors_aux_fini_array_entry
7.    0.00    1.40    0.00                             __frame_dummy_init_array_entry
8.    0.00    1.40    0.00                             __gmon_start__
9.    0.00    1.40    0.00                             __libc_csu_fini
10.   0.00    1.40    0.00                             __libc_csu_init
11.   0.00    1.40    0.00                             _fini
12.   0.00    1.40    0.00                             _init
13.   0.00    1.40    0.00                             _start
14.   0.00    1.40    0.00                             atexit
15.   0.00    1.40    0.00                             call_gmon_start
16.   0.00    1.40    0.00                             data_start
17.   0.00    1.40    0.00                             deregister_tm_clones
18.   0.00    1.40    0.00                             frame_dummy
19.   0.00    1.40    0.00                             register_tm_clones
```

Заметим, что время выполнения функции main указанное в gprof не сильно отличается от указанного после измерений через функцию time. Ну собственно так оно и должно быть. (хотя во втором случае отличается на заметную величину, однако это связанно с каким-то изменениями в нагрузке на систему, потому что time проведённые незамедлительно после gprof указал на почти тот же результат), а результат указанный в самом первом пункте этой работы был проведён достаточно давно на тот момент.

Также хочу отметить, что gprof в отличие от time показывает всё время выполнения процесса, с учётом системных затрат, которые в time указываются через sys.

4. Опыт с rdtsc
    1. Несколько важных комментариев:
        1. Обычно rdtsc даёт оченьхорошую точность вплоть до наносекунд.
        2. Ассемблерная команда rdtsc возвращает количество тиков с момента старта компьютера в паре регистров edx:eax вне зависимости от разрядности операционной системы.
        3. Для того, чтобы перевести в 64-разрядной системе значение в rax можно использовать ассемблерную вставку:

```
shl   rdx, 32     // left shift for 32 bits
```

or rax, rdx // Compose both registers in 64 bit RAX

однако я предпочту сделать это методами си

4. После этого мы будем использовать возможность связывания ассемблера с описанными переменными языка си, таким образом, получим следующий код:

(очень много можно прочитать про ассемблерные вставки по ссылке http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html)

```
unsigned long long int rdtsc(void)
{
   unsigned long long int x;
   unsigned a, d;

   __asm__ volatile("rdtsc" : "=a" (a), "=d" (d));
        // После первого двоеточия указываются операнды вывода, первая «a» означает
регистр eax, вторая «a» – внешнюю переменную объявленную в коде выше, аналогично с «d»

   return ((unsigned long long)a) | (((unsigned long long)d) << 32);
}
```

Именно этот код я буду использовать в замерах.

5. Однако, вообще говоря, существует множество более человечных способов для программиста использования rdtsc:

Под windows например, можно подключить библиотеку intrin.h в которой определена __rdtsc(); или её простая альтернатива getTimeStamp, тело которой выглядит так:

```
__int64 getTimeStamp()
{
  return __rdtsc();
}
```

Пример использования может быть таким:

```
// rdtsc.cpp
// processor: x86, x64
#include <stdio.h>
#include <intrin.h>

#pragma intrinsic(__rdtsc)
// Intrinsic – обязателен

int main()
{
    unsigned __int64 i;
    i = __rdtsc();
    printf_s("%I64d ticks\n", i);
}
Вывод: 3363423610155519 ticks
```

2. Я проводил тесты, используя функцию с ассемблерной вставкой, и получил следующие результаты, которые вполне совпали с другими способами измерения:

number of ticks = 1584771812
number of ticks = 1566574686
number of ticks = 1658164821
number of ticks = 1592471582
number of ticks = 1569720530
number of ticks = 1582264168
number of ticks = 1570760025
number of ticks = 1593575535
number of ticks = 1629494500
number of ticks = 1573569621
среднее = 1592136728
с учётом тактовой частоты процессора это примерно 0,663 секунды,

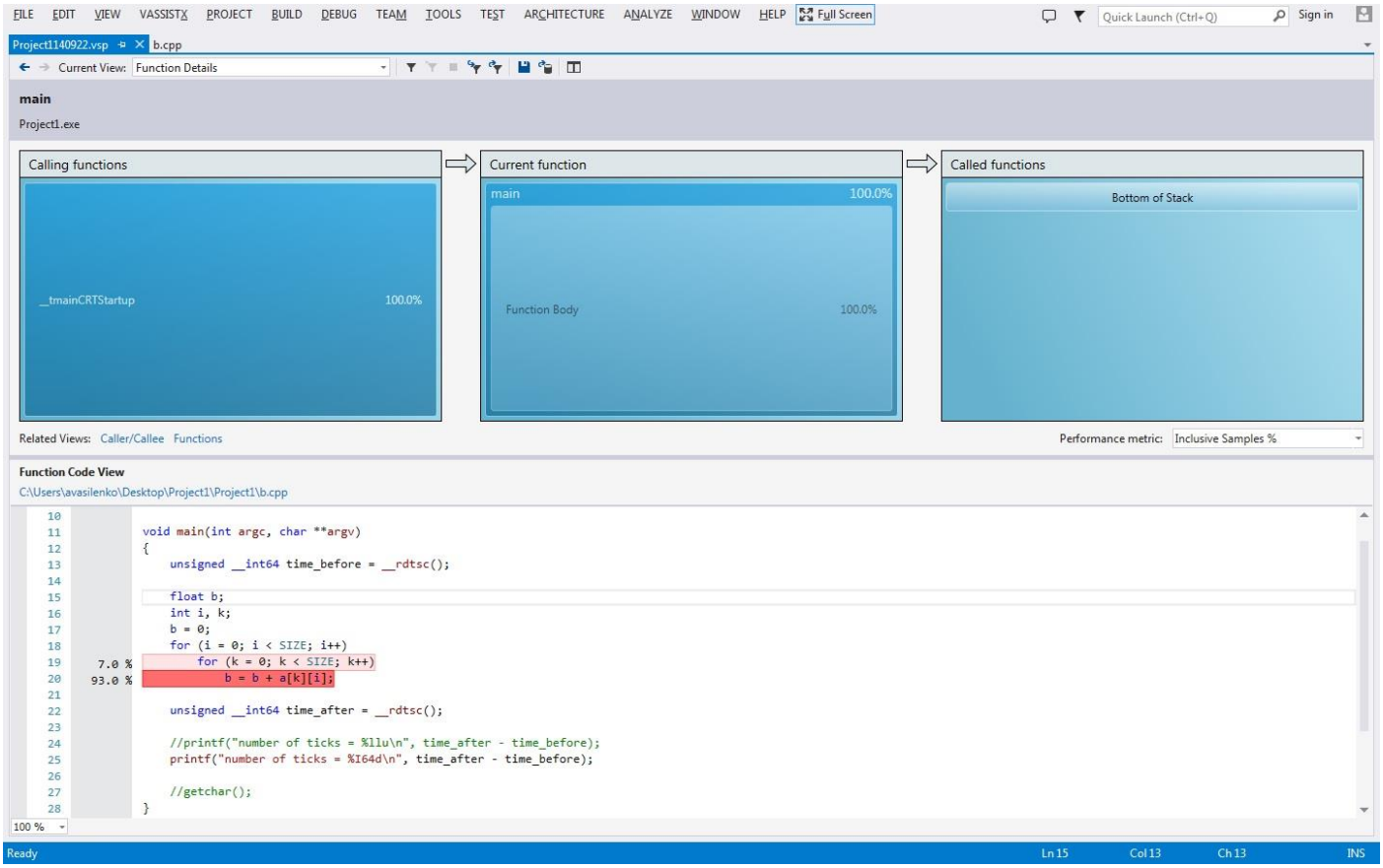
После изменения порядка просмотра массива в цикле (a[i][k] -> a[k][i]):
number of ticks = 3317490683
number of ticks = 3345373438
number of ticks = 3298421271
number of ticks = 3212642912
number of ticks = 3191041134
number of ticks = 3228493816
number of ticks = 3199944640
number of ticks = 3186065623
number of ticks = 3398632487
number of ticks = 3270805276
среднее = 3264891128
с учётом тактовой частоты процессора это примерно 1,36 секунды

Тестирование под windows

1. Характеристики системы:
    1. Процессор - Core 2 Duo - 2 ядра по 2,4 GHz (используется конечно же одно (за исключением того, что некоторые вызовы оси могут быть паралельны))
    2. Windows 7 x64 – хостовая машина (не виртуальная)
2. Компиляция с полным отключением оптимизации, тестирование было через __rdtsc () (смотри пример использования выше)
3. Результаты (компиляция была в режиме Release x64)
   number of ticks = 1272429415
   number of ticks = 1217791503
   number of ticks = 1211113828
   number of ticks = 1217113165
   number of ticks = 1214894971
   number of ticks = 1201390669
   number of ticks = 1199625319
   number of ticks = 1193677596
   number of ticks = 1199743398
   number of ticks = 1209035727
   среднее значение = 1213681559,1
   с учётом тактовой частоты это 0,505s – что несколько быстрее, чем в случаях тестирования из под linux, установленного как виртуальная машина
4. При смене индексации по массиву результаты становятся следующими:
   number of ticks = 3442521213
   number of ticks = 3607482681
   number of ticks = 3339299044
   number of ticks = 3295363860
   number of ticks = 3383916652
   number of ticks = 3339608590
   number of ticks = 3376704114
   number of ticks = 3327883992
   number of ticks = 3320031942
   number of ticks = 3479668713
   среднее значение = 3391248080,1

с учётом тактовой частоты это 1,413s – таким образом прирост от смены системы составил почти ничего, потому что проблема в том, что всё упирается в скорость передачи по шине через кеши.

5. Отмечу любопытный факт – если установить в Visual Studio флаг оптимизации на максимальную производительность (так оно по умолчанию), то количество тактов, за которое будет выполняться программа, будет равно 36 тактам.
6. Использование профилирования в Visual Studio:
   Если пойти в Analyze -> Performace and Diagnostics, то запустится профилирование программы, и потом можно будет посмотреть результаты в виде красивых таблиц, и пометок прямо в коде, вот таких например:



   Здесь прямо указано, на что тратится больше всего времени (в режиме с инвертированными индексами)
   Так же есть таблицы с общим выполнением программы по времени, таблицы с учетом выполнения по времени ассемблерных конструкций, в которые скомпилировался код.

Объяснения (подведение итогов)

1. Способов профилирования – тьма
2. Самое удобное – в Visual Studio
3. Важно учитывать время, затраченное на то, чтобы перегнать данные из оперативной памяти в процессор, поэтому порядок индексирования важен
4. Необходимо использовать несколько замеров, что бы усреднить разброс.
5. Выполнение программ под виртуальной машиной сильно медленнее (0,5s против 1s)
6. Компиляция в режиме оптимизации может сильно сократить время работы (сокращение было с миллиарда до 36 тактов).
7. Для того чтобы уменьшить разброс было запущенно по 10 тестов для time и rdtsc.