Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Дискретный анализ»

Студент: М. В Спиридонов
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2017

Лабораторная работа №3

Задача: Необходимо провести исследование скорости выполнения и потребления оперативной памяти программы, реализованной в лабораторной работе №2.

Вариант дерева: PATRICIA.

Вариант ключа: регистронезависимая последовательность букв английского алфавита длиной не более 256 символов.

Вариант значения: числа от 0 до $2^{64} - 1$.

1 Описание

Для того, чтобы написать качественную программу, нужны утилиты, которые бы тестировали ее по двум основным параметрам: потребление оперативной памяти (а также полное освобождение выделенной программе памяти после завершения программы).

Для первичного поиска «узких» мест я использовал утилиту gprof. А также gcov для более подробного выявления «узких» мест в программе и еще использовал GPT(Google Perfomance Tools), для поиска наиболее медленных мест программы.

Для поиска утечек памяти я использовал valgrind. Немного неприятным был тот факт, что на Mac OS X он не работает как надо, а точнее, из-за особенностей ядра, он указывает на утечки там, где их нет (в некоторых системных библиотеках, например), также чтобы следить за количеством выделяемой памяти я использовал massif, поставляемый вместе с valgrind. Пришлось использовать несколько разных систем, чтобы добиться максимальной работоспособности программы.

1.1 Valgrind

Valgrind имеет модульную архитектуру, и состоит из ядра, которое выполняет эмуляцию процессора, а конкретные модули выполняют сбор и анализ информации, полученной во время выполнения кода на эмуляторе. Valgrind работает под управлением ОС Linux на процессорах x86, amd64, ppc32 и ppc64 (стоит отметить, что ведуться работы по переносу Valgrind и на другие ОС), при этом существуют некоторые ограничения, которые потенциально могут повлиять на работу исследуемых программ. В поставку valgrind входят следующие модули-анализаторы:

- memcheck - основной модуль, обеспечивающий обнаружение утечек памяти, и прочих ошибок, связанных с неправильной работой с областями памяти — чтением или записью за пределами выделенных регионов и т.п.
- cachegrind - анализирует выполнение кода, собирая данные о (не)попаданиях в кэш, и точках перехода (когда процессор неправильно предсказывает ветвление). Эта статистика собирается для всей программы, отдельных функций и строк кода.
- callgrind - анализирует вызовы функций, используя примерно ту же методику, что и модуль cachegrind. Позволяет построить дерево вызовов функций, и соответственно, проанализировать узкие места в работе программы.
- massif - позволяет проанализировать выделение памяти различными частями программы.
- helgrind - анализирует выполняемый код на наличие различных ошибок синхронизации, при использовании многопоточного кода, использующего POSIX Threads.

Если общий префикс есть, но не совпадает с ключом, то поиск также является неудачным.

1.2 Google Performance Tools (GPT)

Google Performance Tools (GPT) — набор утилит, которые позволяют проводить анализ производительности программ, а также анализировать выделение памяти программами и производить поиск утечек памяти.
GPT может работать практически на всех Unix-совместимых операционных системах: Linux, FreeBSD, Solaris, Mac OS X (Darwin), включая поддержку разных процессоров x86, x86_64 и PowerPC. Кроме того, tcmalloc можно скомпилировать также и для MS Windows, что позволит искать утечки памяти в программах, разработанных для этой ОС.

Google Performance Tools состоят из двух библиотек:

- tcmalloc (Thread-Caching Malloc) – очень быстрая реализация malloc. С помощью данной библиотеки можно анализировать выделение памяти в программе, а также производить поиск утечек памяти.
- profiler – данная библиотека реализует анализ производительности выполняемого кода.

Также в пакет GPT входит утилита pprof, для визуализации.

2 Консоль

Для начала проверим реализацию дерева на утечки памяти с помощью утилиты Valgrind и модуля memcheck.

```
$ make
g++ -g -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -c PatriciaTree.cpp
g++ -g -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -c main.cpp
g++ -g -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -o Patricia PatriciaTree.o main.o -lm
$ valgrind ./Patricia < tests/01.t > /dev/null
==28290== Memcheck, a memory error detector
==28290== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==28290== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==28290== Command: ./Patricia
==28290==
==28290==
==28290== HEAP SUMMARY:
==28290==     in use at exit: 0 bytes in 0 blocks
==28290==   total heap usage: 13,632 allocs, 13,632 frees, 1,952,048 bytes allocated
==28290==
==28290== All heap blocks were freed -- no leaks are possible
==28290==
==28290== For counts of detected and suppressed errors, rerun with: -v
==28290== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$ wc tests/01.t
  14905   29810 3951369 tests/01.t
$ du -h tests/01.t
3,8M tests/01.t
```

Теперь с помощью утилиты massif из пакета Valgrind посмотрим на количество памяти, которое выделяется при выполнении программы в зависимости от времени.

```
$ valgrind --tool=massif ./Patricia < tests/01.t > /dev/null
==28371== Massif, a heap profiler
==28371== Copyright (C) 2003-2015, and GNU GPL'd, by Nicholas Nethercote
==28371== Using Valgrind-3.12.0 and LibVEX; rerun with -h for copyright info
==28371== Command: ./Patricia
==28371==
==28371==
$ ms_print massif.out.28371 | head --lines=35
--------------------------------------------------------------------------------
Command:            ./Patricia
Massif arguments:   (none)
ms_print arguments: massif.out.28371
--------------------------------------------------------------------------------


    MB
1.676^                                                                       #
     |                                                                     @@@#
     |                                                                   @@@@@@@#
     |                                                                 @@@@@@@@@@#
     |                                                               @@@@@@@@@@@@@#
     |                                                             @@@@@@@@@@@@@@@@#
     |                                                         @::@ @@@@@@@@@@@@@@@@#
     |                                                    ::::@::@ @@@@@@@@@@@@@@@@#
     |                                               @@@::::@::@ @@@@@@@@@@@@@@@@#
     |                                          ::@::@@@::::@::@ @@@@@@@@@@@@@@@@#
     |                                    @::::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
     |                               @@@@:@: ::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
     |                           @::@ @@:@: ::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
     |                      :@@:@::@ @@:@: ::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
     |                  @::@@:@::@ @@:@: ::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
     |              :@@::@::@@:@::@ @@:@: ::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
     |          @@@:@@: @::@@:@::@ @@:@: ::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
     |      @@@@@@@:@@: @::@@:@::@ @@:@: ::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
     |  ::@@@@@ @@@:@@: @::@@:@::@ @@:@: ::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
     |::::::@ @@@ @@@:@@: @::@@:@::@ @@:@: ::@: @@@::::@::@ @@@@@@@@@@@@@@@@#
   0 +--------------------------------------------------------------------->Mi
     0                                                                   803.5
```

Получив программу, которая не теряет память по время выполнения, я стал задумываться о поиске «узких» мест. Первым делом я использовал стандартный профилеровщик из пакета GCC gprof. Чтобы получить подробную сводку о том, что выполняется в программе дольше всего, для компиляции нужно было добавить флаг -pg. И после этого через утилиту gprof получить flat-профиль.
Также весьма интересен тот факт, что для нормального функционирования gprof'а необходимо было использовать gcc версии <4.9. В противном случае я получал всегда пустой flat-профиль.

```
$ make
g++-4.9 -g -pg -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -c PatriciaTree.cpp
g++-4.9 -g -pg -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -c main.cpp
g++-4.9 -g -pg -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -o Patricia PatriciaTree.o main.o -lm
$ ./Patricia < tests/01.t > /dev/null
$ gprof Patricia gmon.out -p | head --lines=10
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.14      0.01     0.01                             TPatriciaTree::Insert
(char const*, unsigned int, unsigned long long)
  0.00      0.01     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z6LengthPKc
  0.00      0.01     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z7ToLowerPci
  0.00      0.01     0.00        1     0.00     0.00  TPatriciaTree::~TPatriciaTree()
$ wc tests/01.t
  14905   29810 3951369 tests/01.t
```

На 14905 входных данных, состоявших только из вставки удаления и поиска элементов, больше всего времени было потрачено на вставку в дерево. Чтобы точно найти место, где при вставке тратилось больше времени, я использовал gcov. Об этом далее.

Чтобы посмотреть на какие строчки при выполнении программа тратит больше всего времени, я использовал gcov. Чтобы использовать эту утилиту при компиляции нужно указать два флага -ftest-coverage -fprofile-arcs. Использовать gcc версии 4.9 на данном этапе более не обязательно.

```
$ make
g++ -g -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -ftest-coverage -fprofile-arcs -c PatriciaTree.cpp
g++ -g -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -ftest-coverage -fprofile-arcs -c main.cpp
g++ -g -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -ftest-coverage -fprofile-arcs -o Patricia PatriciaTree.o main.o -lm
$./Patricia <tests/01.t > tmp
$ gcov main.cpp
File 'main.cpp'
Lines executed:66.67% of 48
Creating 'main.cpp.gcov'

File '/usr/include/c++/6/iostream'
Lines executed:100.00% of 1
Creating 'iostream.gcov'

File '/usr/include/c++/6/bits/basic_ios.h'
Lines executed:100.00% of 3
Creating 'basic_ios.h.gcov'

File '/usr/include/c++/6/bits/ios_base.h'
Lines executed:100.00% of 1
Creating 'ios_base.h.gcov'

File '/usr/include/c++/6/istream'
Lines executed:100.00% of 1
Creating 'istream.gcov'

File '/usr/include/c++/6/ostream'
Lines executed:100.00% of 4
Creating 'ostream.gcov'
```

```
$ cat main.cpp.gcov
        -:    0:Source:main.cpp
        -:    0:Graph:main.gcno
        -:    0:Data:main.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <iostream>
        -:    2:#include <fstream>
        -:    3:#include "PatriciaTree.h"
        -:    4:#include <stdlib.h>
        -:    5://#include "dmalloc.h"
        -:    6:
    #####:    7:void ToLower (char *t, int len)
  7646265:    8:    for (int i = 0; i < len; ++i)
  3815680:    9:        if (t[i] >= 'A' && t[i] <= 'Z')
  3163648:   10:            t[i] = 'a' + t[i] - 'A';
        -:   11:
        -:   12:
    #####:   13:
        -:   14:
    #####:   15:unsigned int LengthString (const char *string)
    #####:   16:    unsigned int length = 0;
        -:   17:
  3830585:   18:    while ((string + length != nullptr) && string[length] != '\0')
  3815680:   19:        ++length;
        -:   20:
        -:   21:
    #####:   22:    return length;
        -:   23:
        -:   24:
        1:   25:int main ()
        -:   26:    //char *pt = (char*) malloc(1023);
        -:   27:    //pt[1023] = '\0';
        -:   28:   // std::ios::sync_with_stdio(false); //make_faster
        -:   29:    char choice;
        2:   30:    if (!std::cin.get(choice))
        -:   31:        return 0;
        -:   32:
        1:   33:    TPatriciaTree *p = new TPatriciaTree;
        1:   34:    const int size = 1025; //1024+1
        -:   35:    char temp[size];
```

```
    -:   36:
    -:   37:    unsigned long long data;
    -:   38:
    -:   39:    int len;
    -:   40:
    -:   41:
    -:   42:
    1:   43:    std::cin.putback(choice);
    -:   44:
29812:   45:    while (std::cin >> choice)
14905:   46:        switch (choice)
    -:   47:            case '+':
 4947:   48:                std::cin >> temp >> data;
 4947:   49:                len = LengthString(temp);
 4947:   50:                ToLower(temp, len);
    -:   51:
 4947:   52:                if (p->Insert(temp, len, data))
 4947:   53:                    std::cout << "OK\ n";
    -:   54:                 else
#####:   55:                    std::cout << "Exist\ n";
    -:   56:
    -:   57:
    -:   58:                break;
    -:   59:            case '-':
 5011:   60:                std::cin >> temp;
 5011:   61:                len = LengthString(temp);
 5011:   62:                ToLower(temp, len);
    -:   63:
 5011:   64:                if (p->Remove(temp, len))
#####:   65:                    std::cout << "OK\ n";
    -:   66:                 else
 5011:   67:                    std::cout << "NoSuchWord\ n";
    -:   68:
    -:   69:
    -:   70:                break;
    -:   71:            case '!':
#####:   72:                std::cin.get(choice);
#####:   73:                std::cin.get(choice);
    -:   74:
#####:   75:                if (choice == 'S')
#####:   76:                    std::cin >> temp >> temp;
```

```
    -:   77:
#####:   78:                    p->Save(temp);
    -:   79:                 else
#####:   80:                    std::cin >> temp >> temp;
#####:   81:                    delete p;
    -:   82:
#####:   83:                    p = new TPatriciaTree;
#####:   84:                    p->Load(temp);
    -:   85:
    -:   86:
    -:   87:
    -:   88:                break;
    -:   89:            default:
    -:   90:                int len;
 4947:   91:                std::cin.putback(choice);
 4947:   92:                std::cin >> temp;
    -:   93:
 4947:   94:                len = LengthString(temp);
 4947:   95:                ToLower(temp, len);
 4947:   96:                const unsigned long long *d = p->Search(temp, len);
    -:   97:
 4947:   98:                if (d == nullptr)
 2400:   99:                    std::cout << "NoSuchWord\ n";
    -:  100:                 else
 7641:  101:                    std::cout << "OK: " << *d << '\ n';
    -:  102:                    d= nullptr;
    -:  103:
    -:  104:
    -:  105:
    -:  106:
    -:  107:
    -:  108:
    1:  109:    delete p;
    -:  110:    return 0;
    2:  111:
```

Из информации, выведенной gcov'ов можно точно утверждать, что на данном тесте ровно 4947 раз производилась вставка и 5011 удаление, а также 4947 раз производился поиск. Т.е. почти все время, которое работала программа, заняла вставка 4947 элементов. Стоит посмотреть на каких строчках происходила такая задержка и почему. Выведу только результаты профилирования метода вставки.

```
$ gcov PatriciaTree.cpp
...
$ cat PatriciaTree.cpp.gcov
...
     4947:   62:bool TPatriciaTree::Insert(const char *t, unsigned int len,
      unsigned long long num)
     4947:   63:    TPatriciaTree *node = this;
     4947:   64:    TPatriciaTree *parent = nullptr;
        -:   65:    int prefix;
     4947:   66:    bool sonsBrother = false;
     4947:   67:    ++len;
     4947:   68:    if (node->length == 0)
        1:   69:        node->key = new char[len];
        2:   70:        CopyStr(node->key, t, len);
        1:   71:        node->length = len;
        1:   72:        node->data = num;
        1:   73:        return true;
        -:   74:
        -:   75:
        -:   76:    while (true)
   149220:   77:        if (len == 0)
    #####:   78:            ++len;
        -:   79:
   149220:   80:        if (node == nullptr)
     9892:   81:            node = new TPatriciaTree;
        -:   82:            delete []  node->key;
     4946:   83:            node->key = new char[len];
     9892:   84:            CopyStr(node->key, t, len);
     4946:   85:            node->length = len;
     4946:   86:            node->data = num;
        -:   87:
     4946:   88:            if (parent != nullptr)
     4946:   89:                if (sonsBrother)
     4946:   90:                    parent->next = node;
        -:   91:                 else
    #####:   92:                    node->next = parent->link;
    #####:   93:                    parent->link = node;
        -:   94:
                -:   95:
        -:   96:            break;
        -:   97:
```

```
     -:   98:
288548:   99:        prefix = Prefix(node->key, node->length, t, len);
     -:  100:
144274:  101:        if (prefix == 0)
134493:  102:            sonsBrother = true;
134493:  103:            parent = node;
134493:  104:            node = node->next;
  9781:  105:         else if (prefix < len)
  9781:  106:            if (prefix < node->length)
  2490:  107:                TPatriciaTree *newNode = new TPatriciaTree;
  1245:  108:                newNode->key = new char[node->length - prefix];
  2490:  109:                CopyStr(newNode->key, node->key + prefix,
   node->length - prefix);
  1245:  110:                newNode->length = node->length - prefix;
  1245:  111:                newNode->data = node->data;
  1245:  112:                newNode->link = node->link;
  1245:  113:                node->link = newNode;
  1245:  114:                char *buf = new char[prefix];
  2490:  115:                CopyStr(buf, node->key, prefix);
  1245:  116:                delete[] node->key;
  1245:  117:                node->key = buf;
  1245:  118:                node->length = prefix;
     -:  119:
  9781:  120:            sonsBrother = false;
  9781:  121:            parent = node;
  9781:  122:            node = node->link;
  9781:  123:            t += prefix;
  9781:  124:            len -= prefix;
     -:  125:         else
     -:  126:            return false;
     -:  127:
     -:  128:
     -:  129:    return true;
     -:  130:
```

Результаты профилирования метода вставки говорят о том, что 9892 раз было создано новое дерево (т.к. оно самоподобно каждый новый узел является полноценным деревом) и 288548 раз был посчитан префикс двух строк. Также весьма сомнительно выглядит 9892 вызовов метода копирования части каких-то ключей. Судя по информации выше, можно точно сказать что это и есть три самые долгие по времени выполнения строчки. Были предприняты меры по оптимизации.

После предпринятых мер по оптимизации программы, я решил сверить результаты использованных ранее утилит с результатами, полученными от Google Perfomance Tools. Первым делом я хотел бы проверить программу на возможные утечки памяти, а потом уже на производительность и наличие «узких» мест.

```
$ make
g++ -g -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -c PatriciaTree.cpp
g++ -g -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -c main.cpp
g++ -g -std=c++11 -pedantic -Wall -Wno-sign-compare -Wno-long-long
-O2 -o Patricia PatriciaTree.o main.o -lm
$ LD_PRELOAD=/usr/local/lib/libtcmalloc.so.4.4.5 HEAPCHECK=normal
./Patricia < tests/01.t > /dev/null
WARNING: Perftools heap leak checker is active -- Performance may suffer
Have memory regions w/o callers: might report false leaks
No leaks found for check "_main_" (but no 100
% guarantee that there aren't any): found 16 reachable heap objects of 81386 bytes
$ LD_PRELOAD=/usr/local/lib/libprofiler.so.0.4.14 CPUPROFILE=profile.o
 CPUPROFILE_FREQUENCY=100000 ./Patricia < tests/01.t > /dev/null
PROFILE: interrupts/evictions/bytes = 29/13/1640
$ pprof --text Patricia profile.o
Using local file Patricia.
Using local file profile.o.
Total: 29 samples
       8  27.6%  27.6%       25  86.2% std::operator>>
       7  24.1%  51.7%        7  24.1% _IO_getc
       4  13.8%  65.5%        4  13.8% _IO_acquire_lock_fct
       4  13.8%  79.3%        6  20.7% _IO_ungetc
       2   6.9%  86.2%        2   6.9% Prefix (inline)
       2   6.9%  93.1%        2   6.9% __GI__IO_sputbackc
       1   3.4%  96.6%        1   3.4% LengthString (inline)
       1   3.4% 100.0%        4  13.8% __gnu_cxx::stdio_sync_filebuf::uflow
       0   0.0% 100.0%        2   6.9% TPatriciaTree::Remove
       0   0.0% 100.0%        6  20.7% __gnu_cxx::stdio_sync_filebuf::underflow
       0   0.0% 100.0%       29 100.0% __libc_start_main
       0   0.0% 100.0%       29 100.0% _start
       0   0.0% 100.0%       29 100.0% main
       0   0.0% 100.0%        1   3.4% std::istream::_M_extract
       0   0.0% 100.0%        1   3.4% std::istream::operator>> (inline)
       0   0.0% 100.0%        1   3.4% std::istream::sentry::sentry
```

3 Выводы

Профилирование и тестирование заняло намного больше времени, чем написание самой программы. По приблизительным подсчётам раз так в пять. Но хочу сказать, что профилирование мне понравилось намного больше. Я получил много информации о том, как и почему может тормозить программа и как точно найти место, где она течёт по памяти или долго выполняется. Огромное удовольствие доставила библиотека Google Perfomance Tools. Хоть и пытался поставить и запустить ее я более чем три дня, итог меня приятно удивил. Я познал уйму особенностей разных операционных систем и разобрался почему, например выводы утилит профилирования в зависимости от системы иногда показывает совершенно разные результаты. Было интересно разбираться, например, почему в Mac OS X GTP выдает странные результаты время от времени, но все свелось к особенностям работы самой операционки. Также я разобрался как запустить все предложенные на лекциях средства профилирования, но включил в отчет отнюдь не все, потому что функционал схож. Например, использование библиотеки dmallocxx (реализация dmalloc для c++) почти не имеет смысла при наличие Valgrind'a. Также очень понравилось визуализировать вывод GPT и Gprof при помощи graphviz и pprof, но для себя я выяснил, что в принципе никому красивые графики/диаграммы/графы не нужны, разве что для презентации какой-нибудь или огромного проекта с миллионами функций, разве что. Но никак не для лабораторных работ размером меньше двух тысяч строк кода.

Список литературы

[1] *GitHub одного из семинаристов.*
URL: `https://github.com/toshunster/da`.