**Санкт-Петербургский политехнический университет Петра Великого**

**Кафедра компьютерных систем и программных технологий**

**Отчет по дисциплине**

**«Проектирование ОС и их компонентов»**

**Профилирование программ (C/C++)**

**под Windows/Linux**

**Работу выполнил студент группы №:** 13541/3 Чеботарёв М. М.

**Работу принял преподаватель:** \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Душутина Е. В.

**Санкт-Петербург**

**2017 г.**

Используемая система и версия ядра

a) Windows

Процессор: Intel(R) Core(TM) i5-2450 CPU @2.50GHz 2.50GHz

ОЗУ: 8,00 Гб

Тип системы: Windows 7 Ultimate Compact (2009) Service Pack 1. x64.

б) Linux

michael@michael-LIFEBOOK-AH531:~$ **lsb\_release -a**

No LSB modules are available.

Distributor ID: Ubuntu

Description: Ubuntu 16.04.1 LTS

Release: 16.04

Codename: xenial

michael@michael-LIFEBOOK-AH531:~$ **cat /proc/version**

Linux version 4.4.0-38-generic (buildd@lgw01-58) (gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.2) ) #57-Ubuntu SMP Tue Sep 6 15:42:33 UTC 2016

**1. МЕТОДЫ ПРОФИЛИРОВАНИЯ**

Профилирование позволяет оценить производительность программы и решить основные проблемы:

* Интенсивное использование ЦП;
* Не самое эффективное использование подсистемы ввода/вывода;
* Уровневое взаимодействие
* Выделение и использование памяти (особенно .Net)
* Излишняя/неправильная синхронизация, недостаточное использование ядер процессора;

**1.1. Sampling (метод Выборки) [1]**

Метод собирает статистические данные о работе приложения (во время профилирования). Этот метод **легковесный** и поэтому, в результате его работы очень маленькая погрешность в полученных данных.

**Каждый определенный интервал времени собирается информация о стеке вызовов** (call stack). На основе этих данные производится подсчет производительности. Используется для первоначального профилирования и **для определения проблем связанных с использованием ЦП.**

**1.2. Instrumentation**

Метод **собирает детализированную информацию о времени работы** каждой вызванной функции. Используется для замера производительности операций **ввода/вывода.**

Метод внедряет свой код в двоичный файл, который фиксирует информацию о синхронизации (времени) для каждой функции в файл, и для каждой функции которые вызываются в этой.

Отчет содержит 4 значения для предоставления затраченного времени:

* **Elapsed Inclusive** — общее время, затраченное на выполнение функции
* **Application Inclusive** — время, затраченное на выполнение функции, за исключением времени обращений к операционной системе.
* **Elapsed Exclusive** — время, затраченное на выполнение кода в теле. Время, которое тратят функции, вызванные целевой функцией.
* **Application Exclusive** — время, затраченное на выполнение кода в теле. Исключается время, которое тратится выполнения вызовов операционной системы и время, затраченное на выполнение функций, вызванные целевой функцией.

**1.3. Concurrency**

Метод собирает информацию **о многопоточных** приложениях. Метод собирает подробную информацию о стеке вызовов, каждый раз, **когда конкурирующие потоки вынуждены ждать** доступа к ресурсу.

**1.4. .NET Memory**

Профайлер собирает **информацию о типе, размере**, а также **количество объектов**, которые были **созданы** в распределении **или были уничтожены** сборщиком мусора. Профилирование памяти почти не влияет на производительность приложения в целом.

**1.5. Tier Interaction**

Метод добавляет информацию в файл для профилирования о синхронных вызовах **ADO.NET** между страницей **ASP.NET** или другими приложениями и **SQL** сервера. Данные включают число и время вызовов, а также максимальное и минимальное время.

**2. ПРОФИЛИРОВАНИЕ СРЕДСТВАМИ Microsoft Visual Studio 2013 & 2017**

**Visual Studio Profiling Tool** позволяет разработчикам измерять, оценивать производительность приложения и кода. Эти инструменты полностью встроены в IDE, чтобы предоставить разработчику беспрерывный контроль.

В данной работе рассматривается профилирование приложения, используя **Sampling** и **Instrumentation** методы профилирования, чтобы выявить проблемы в производительности приложения.

Перед началом работы необходимо произвести подготовку, выполнив следующие шаги:

1. Запустить среду от имени администратора;
2. Установить активную сборку конфигурации (active build configuration) в значение «release» (в меню **Build** выбрать **Configuration Manager** и в поле **Active solution** configurations выбрать **Release**);
3. Установить/проверить обновление символьных файлов [2].

**2.1. Создание и запуск сессии профилирования**

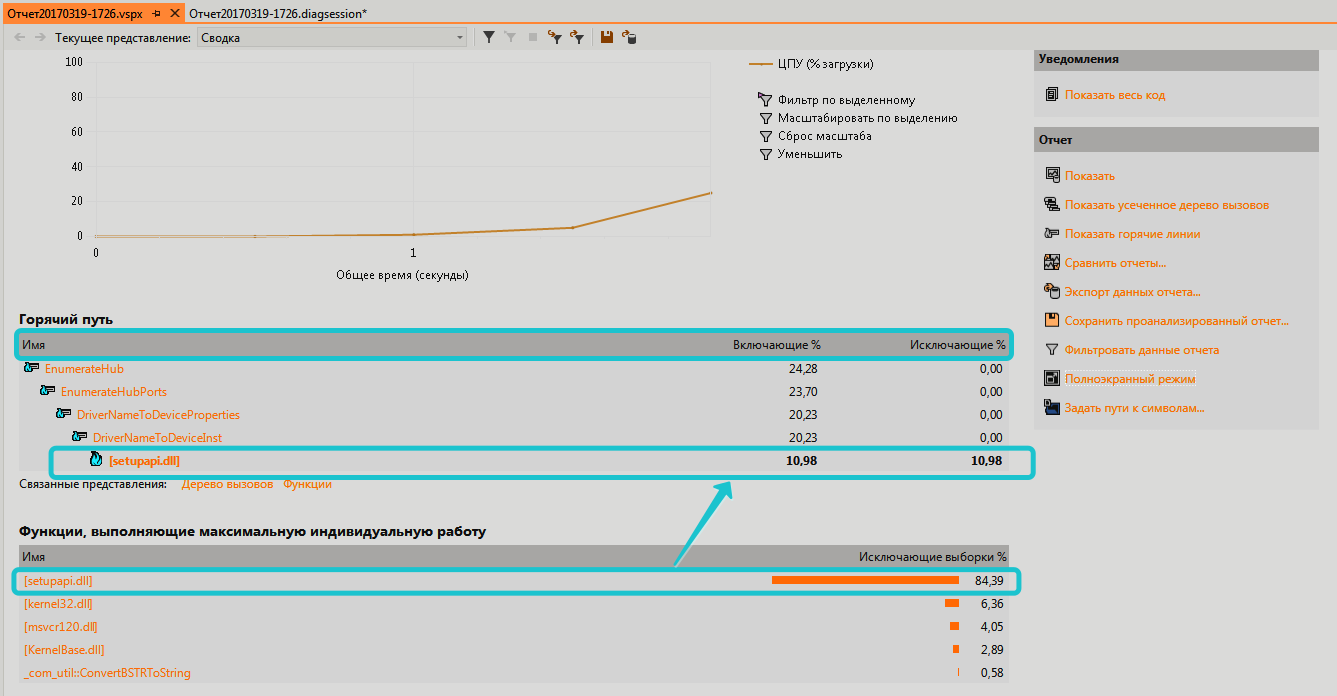
Для того, чтобы включить профилирования для текущего проекта или .exe-файла следует выполнить следующие шаги:

1. Открыть проект;
2. В пункте меню **Analyze** выбрать **Performance and Diagnostic**;
3. Оставить галочку напротив **Performance Wizard**;
4. Нажать кнопку **Start**;
5. Выбрать метод профилирования, например **CPU sampling** (описание методов приводилось выше). Нажать кнопку **Next**;
6. Указать открытый проект, как цель анализа, и нажать **Next**;
7. Оставить галочку «**Launch profiling** …» и нажать **Finish**.

После этого будет выполнен запуск программы, а по ее завершению проведен результат выполнения профилирования. Профайлер сгенерирует отчет (**рис.1**).

|  |
| --- |
|  |

**Рис.1**.а Отчет Sample-профилирования (MVS2013)



**Рис.2**.б Отчет Sample-профилирования (**MVS2017)**

##### 2.2. Анализ отчета Sampling метода (или метода выборки)

В **Summary** отображается график использования процессора в течение всего времени профилирования. В данном случае видно, что нагрузка на ЦП не столь велика и критический момент достигала 20-25%.

Список **Hot Path** показывает ветки вызовов, которые проявили наибольшую активность. Процентное отношение напротив каждой из функций показывает отношение количества зафиксированных вызовов данной функции к общему количеству зафиксированных вызовов.

***Примечание***: метод выборки – статистический метод, и производит запись о исполняемой в данный момент функции каждые N тактов. Таким образом, если функция не успеет выполнится за N тактов, то она будет «зафиксирована» профилировщиком дважды (и более) раз.

В списке **Functions Doing Most Individual Work** – функции, которые занимали б**о**льшее время процесса в теле этих функций.

В данном случае **вызовы** большинства функций **происходят равномерно**, приблизительно одинаковое количество раз.

Если кликнуть на метод, указанный в Hot Path, можно перейти к просмотру стека всех вызовов, совершенных программой. Для корректного отображения вызовов обязательно выполнение шага №3 на подготовительном этапе.

Перемещаясь по стеку вызовов, можно увидеть, процентное отношение длительности выполнения каждого метода ко времени выполнения всей программы. На рисунке 2 приведен пример «разбора» функции **main**, включающей в себя 4 вызова printHostControllersInfo(), printHubsInfo(), printDevicesInfo и system("pause").

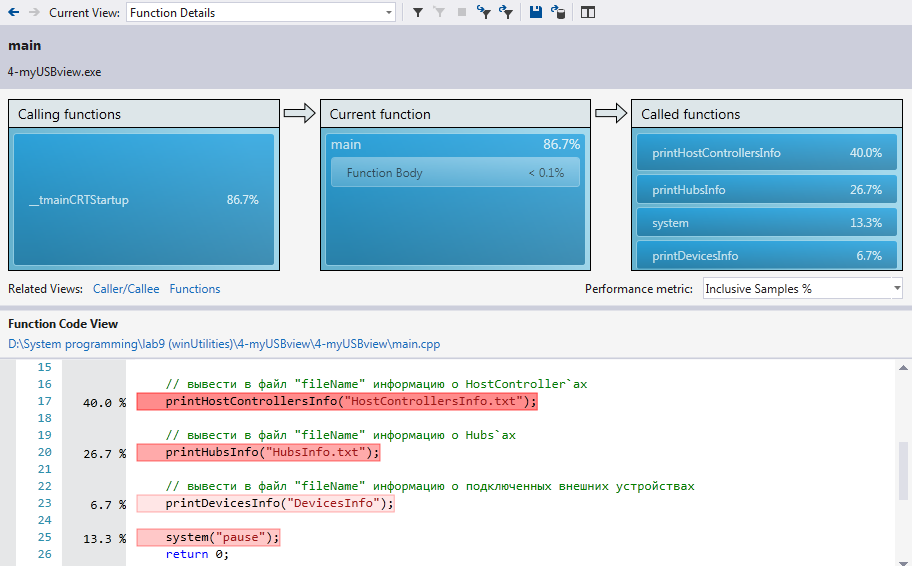


Рис.3. Стек всех вызовов, с указанием загрузки CPU (MVS2013)

Из рисунка видно, что метод сбора сведения о HostController`ах наиболее тяжеловесный и занял 40% всего времени работы программы; второе место, 26.7% занимает время сбора сведений о hub`ах и т.д.

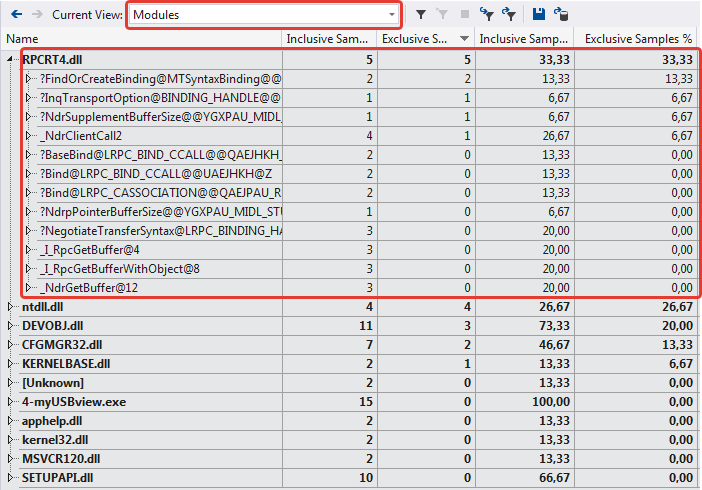


Рисунок 4. Список используемых модулей (MVS2013)

В списке используемых модулей отображаются модули (в основном динамические библиотеки), используемые программой, с указанием используемых ресурсов.

**Сравнение результатов профилирования ДО и ПОСЛЕ оптимизации**

Так как в моем случае наибольшую трудоемкость составляют системные вызовы обращения к устройствам и реестру Windows, оптимизировать данную часть крайне затруднительно, поэтому вместо этого облегчим программу отключив часть функциональности. Удвоим самый ресурсоемкий вызов – printHostControllersInfo() (рис.4.), и повторим Sample-профилирование (рис.5.):

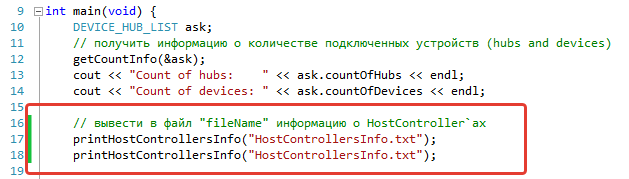


Рисунок 5. Добавили лишний метод (MVS2013)

Запускаем профилирование. Для того, чтобы сравнить результаты работы ДО и ПОСЛЕ, следует в меню **Performance Explorer** выбрать оба отчета, кликнуть ПКМ и выбрать **Compare Performance Reports.** В результате (рис.5) мы увидим отражение изменений в сводной таблице: здесь приведены значения обоих отчетов профилирования и их разница.

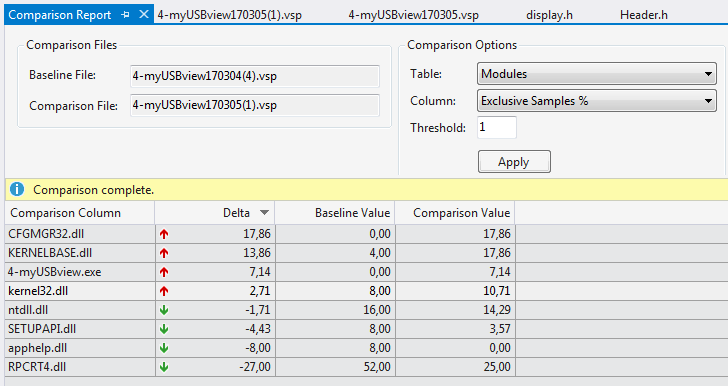


Рисунок 6. Сравнение отчетов профилирования (MVS2013)

Однако, если запустить одну и ту же программу несколько раз, значения методом Sample могут очень сильно отличаться (рис.6). Поэтому рассмотрим еще несколько методов.

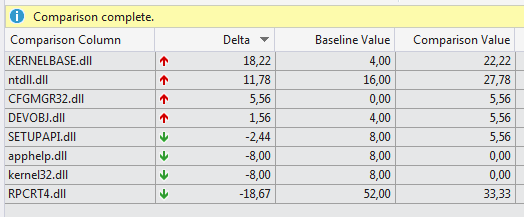


Рисунок 7. Сравнение отчетов ОДИНАКОВЫХ программ (MVS2013)

**ВЫВОД**: **метод выборки не подходит** для анализа производительности данной программы: нагрузка на ЦП не велика, а сама программа проводит большое количество времени при обращении к периферийным устройствам. То есть получить качественную оценку **может помочь метод, учитывающий взаимодействие с устройствами ввода/вывода.**

#### 2.3. Профилирование методом Instrumentation

Этот метод полезен при профилировании операций ввода вывода, запись на диск и при обмене данными по сети. Этот метод предоставляет **больше информации**, чем предыдущий, но он несет с собой **больше накладных расходов**, понятно, что **«чем больше сила, тем больше ответственность»**. Бинарные файлы, полученные после вставки дополнительного кода получаются больше обычных, и не предназначены для развертывания.

Переключение режима профилирования происходит в меню **Performance Explorer,** нужно кликнуть ПКМ на название, далее **Properties/General/Instrumentation** и **ОК.**

**Примечание**: вызов system(“pause”) был закомментирован, т.к. данный метод вносит серьезные «шумы» в результаты отчета.

Результаты отчета профилирования Instrumentation больше ориентированы на устройства ввода/вывода информации, как то, чем так богата исследуемая программа. Фрагмент отчета приведен на рис.7.

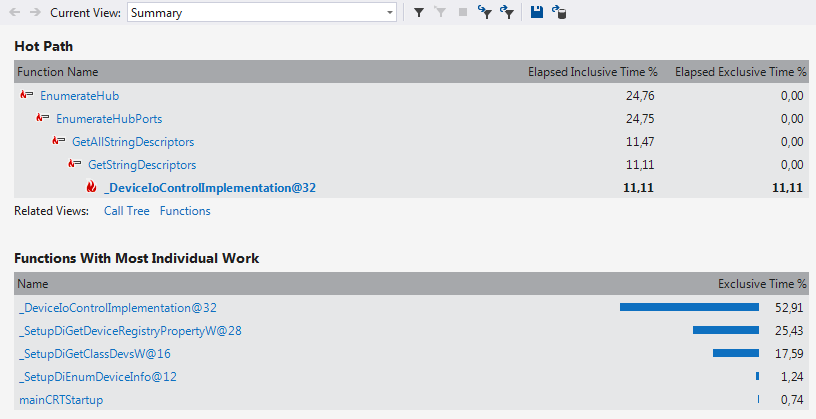


Рисунок 8. Наиболее «тяжелая» последовательность вызовов и самые ресурсоемкие вызовы (MVS2013)

По результатам профилировщика видно, что почти 53% процессорного времени занимают системные вызовы IOCTL, 2-4 место по нагрузке занимают вызовы, обращающиеся к реестру системы.

***Важно примечание***: в анализируемой программе почти **30 IOCTL-вызовов.**

Еще одно важно замечание, **если сравнить несколько отчетов, выполненных методов Instrumentation, то видно, что результаты почти не отличаются**, в отличии от 1го метода.

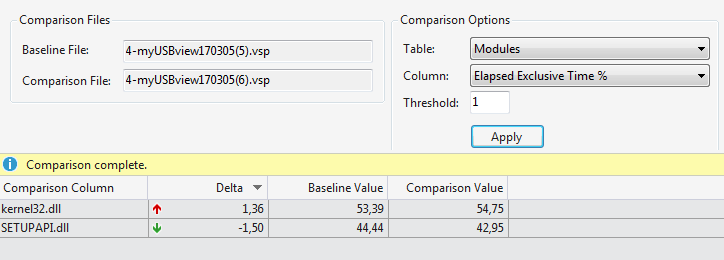


Рисунок 9. Сравнение отчетов методом Instrumentation

#### 2.4. Профилирование использованной памяти в MVS2017

#### Новая (на 2017 год) MVS2017 имеет (выделенный в отдельный) метод оценки использованной памяти, основная задача которого, схожа с задачей программы Dr. Memory – исследование использование памяти на наличие утечек.

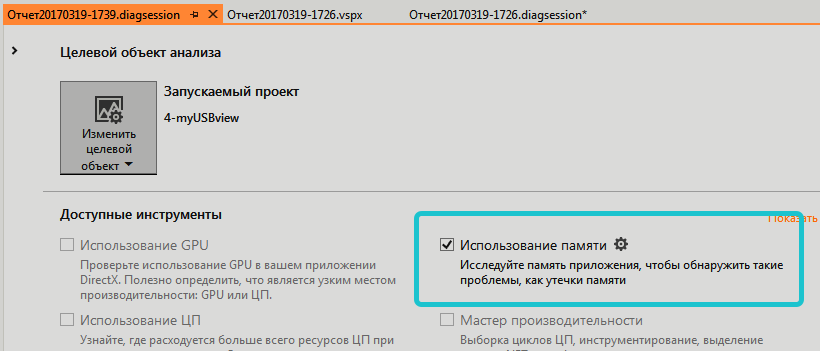
****

Рисунок 10. Настройка профилировщика на анализ использования памяти

**Выполним оба анализа и сравним результат:**

**MVS 2017.** У данного метода профилирования, есть особенность: сохранение данных о текущем состоянии памяти делается вручную, процесс (в данной версии) автоматизировать точно нельзя. При запуске профилировщика памяти будет вызвана анализируемая программа, в ходе ее исполнения необходимо вручную сделать несколько snapshot`ов наиболее интересующих моментов. Т.к. моя программа выполняется менее чем за 1 сек, и просто не получается успеть создать snapshot памяти кучи, я добавил между наиболее массивными вызовами system(«pause»). На выделенную память это не должно повлиять, а вот время сделать «фотографию» кучи даст. Таким образом сделан следующий эксперимент: сделано 5 snapshot`ов, каждый из которых следует от самого начала программы, вплоть до завершения (рис 11).

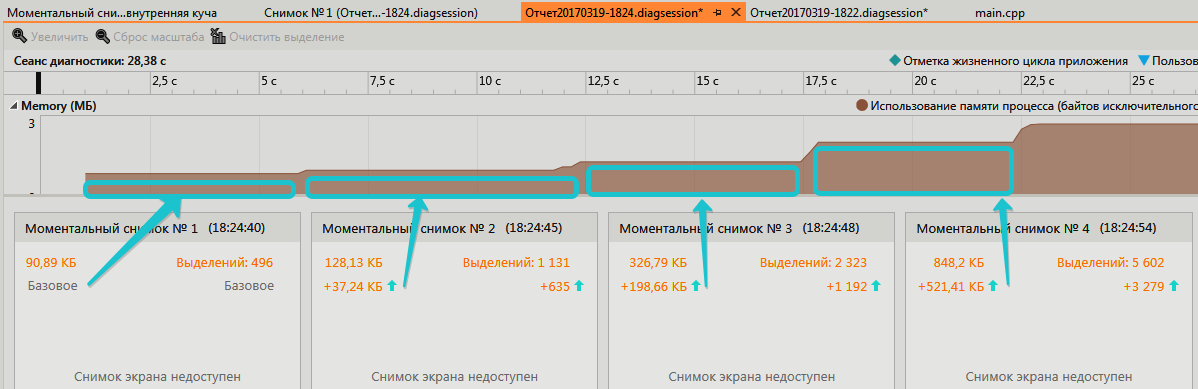
****

Рисунок 11. Анализ использования памяти

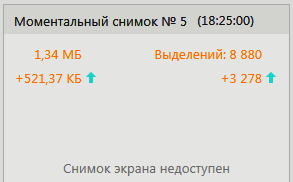
****

Рисунок 12. Анализ использования памяти перед завершением программы

**Итог:** если верить профилировщику по выделению памяти, утечка памяти в моей программе составляет 1.34 МБ, что не делает мне, как программисту, никакой чести.

В защиту своей репутации могу сказать, что в коде самой программы **производится 62 вызова выделения** памяти ALLOC (не учитывая, что вызова буду исполнятся в циклах), и **95 вызовов освобождения** памяти FREE (не учитывая циклические вызовы и что некоторые вызовы исключают друг друга). **Могло быть и хуже** (было хуже).

Для того, чтобы оценить данного профайлера, напишем простую, легкую в понимании программу.

**Тестовая программа по выделению памяти.** Написана небольшая программа, которая выделяет 1Гб памяти, а затем его освобождает.

**HelloWorld.cpp (да, я ее так назвал)**

#include <iostream>

#include <cstdlib> // для system

using namespace std;

int main(void)

{

const int blockCount = 1024;

const int blockSize = 1024 \* 1024;

char \*\*buf;

printf("Hit something...\n");

system("pause");

buf = (char\*\*)malloc(blockCount \* sizeof(char\*));

for (int i = 0; i<blockCount; i++)

{

buf[i] = (char\*)malloc(blockSize \* sizeof(char));

}

printf("Memory allocated\n");

printf("Hit something...\n");

system("pause");

system("pause");

getchar();

for (int i = 0; i<blockCount; i++)

{

free(buf[i]);

}

free(buf);

printf("Hit something...\n");

printf("Memory freed\n");

system("pause");

return 0;

}

Анализ работы с памятью:

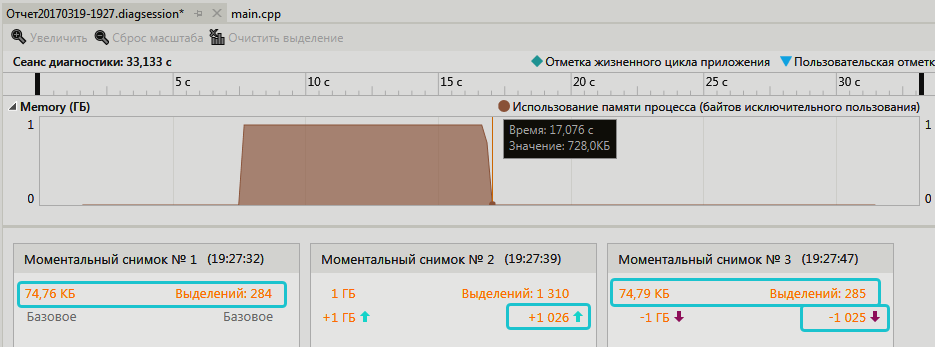


Рисунок 13. Исследование памяти для эталонное программы

**Итог**: в эталонной программе выделено 1ГБ, а освобождено почти 1ГБ: утечка памяти составляет 0,03Кб. И такой результат повторялся при повторных экспериментах, и после перезагрузки ПК. Возможные причины утечки:

* большой объем выделяемой памяти, сразу 1ГБ;
* ошибка в самом профилировщике, точнее MVS 2017;
* ошибка в системной функции malloc или free;
* ошибка в системе Win 7;

#### 2.5. Профилирование в DR. Memory

#### 2.5.1. Используемая память

#### DR. Memory – средство для оценивания работы программ с точки зрения работы с памятью. Проводятся проверки на утечки памяти, неправильную адресацию, выход за пределы памяти и другие ошибки, связанные с памятью.

#### *Примечание*: Dr. Memory существенно увеличивает время исполнения программы, т.к. производится анализ всех системных вызовов.

#### Результат для эталонной программы:

#### Dr. Memory version 1.11.0 build 2 built on Aug 29 2016 02:42:07

#### Dr. Memory results for pid 3580: "Hello World.exe"

#### Application cmdline: ""C:\Users\michael\Desktop\Hello World.exe""

#### Recorded 115 suppression(s) from default C:\Program Files (x86)\Dr. Memory\bin\suppress-default.txt

#### ===========================================================================

#### FINAL SUMMARY:

#### DUPLICATE ERROR COUNTS:

#### SUPPRESSIONS USED:

#### NO ERRORS FOUND:

#### 0 unique, 0 total unaddressable access(es)

#### 0 unique, 0 total uninitialized access(es)

#### 0 unique, 0 total invalid heap argument(s)

#### 0 unique, 0 total GDI usage error(s)

#### 0 unique, 0 total handle leak(s)

#### 0 unique, 0 total warning(s)

#### 0 unique, 0 total, 0 byte(s) of leak(s)

#### 0 unique, 0 total, 0 byte(s) of possible leak(s)

#### ERRORS IGNORED:

#### 6 unique, 9 total, 1608 byte(s) of still-reachable allocation(s)

#### (re-run with "-show\_reachable" for details)

#### Details: C:\Users\michael\AppData\Roaming\Dr. Memory\DrMemory-Hello World.exe.3580.000\results.txt

**Итог: утечки нет**, но 1608 байт не были стерты и все еще могут быть прочитаны другими процессами.

#### Результат для разработанной утилиты:

Dr. Memory version 1.11.0 build 2 built on Aug 29 2016 02:42:07

Dr. Memory results for pid 1020: "4-myUSBview.exe"

Application cmdline: ""D:\System programming\lab9 (winUtilities)\4-myUSBview\Release\4-myUSBview.exe""

Recorded 115 suppression(s) from default C:\Program Files (x86)\Dr. Memory\bin\suppress-default.txt

**Error #1: UNADDRESSABLE ACCESS beyond heap bounds: reading 0x024311fc-0x024311fe 2** byte(s)

# 0 StringCbLengthW [c:\program files (x86)\windows kits\8.1\include\shared\strsafe.h:9925]

# 1 EnumerateHostController [d:\system programming\lab9 (winutilities)\4-myusbview\4-myusbview\hostcontroller.h:242]

# 2 EnumerateHostControllers [d:\system programming\lab9 (winutilities)\4-myusbview\4-myusbview\hostcontroller.h:153]

# 3 printHostControllersInfo [d:\system programming\lab9 (winutilities)\4-myusbview\4-myusbview\mywinusb.h:58]

# 4 \_\_tmainCRTStartup [f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c:626]

...

**Error #124: LEAK 52 direct bytes 0x02545e38-0x02545e6c + 782 indirect bytes**

**# 0 replace\_RtlAllocateHeap** [d:\drmemory\_package\common\alloc\_replace.c:3770]

# 1 KERNELBASE.dll!GlobalAlloc +0x6d (0x76b04e55 <KERNELBASE.dll+0x14e55>)

# 2 EnumerateHub [d:\system programming\lab9 (winutilities)\4-myusbview\4-myusbview\hub.h:136]

# 3 EnumerateHubPorts [d:\system programming\lab9 (winutilities)\4-myusbview\4-myusbview\hub.h:739]

# 4 EnumerateHub [d:\system programming\lab9 (winutilities)\4-myusbview\4-myusbview\hub.h:342]

# 5 EnumerateHostController [d:\system programming\lab9 (winutilities)\4-myusbview\4-myusbview\hostcontroller.h:360]

# 6 EnumerateHostControllers [d:\system programming\lab9 (winutilities)\4-myusbview\4-myusbview\hostcontroller.h:153]

# 7 printDevicesInfo [d:\system programming\lab9 (winutilities)\4-myusbview\4-myusbview\mywinusb.h:100]

# 8 \_\_tmainCRTStartup [f:\dd\vctools\crt\crtw32\dllstuff\crtexe.c:626]

# 9 KERNEL32.dll!BaseThreadInitThunk +0x11 (0x764b33aa <KERNEL32.dll+0x133aa>)

===========================================================================

FINAL SUMMARY:

**DUPLICATE ERROR COUNTS:**

**Error # 1: 3**

**...**

**Error # 123: 2**

**Error # 124: 2**

SUPPRESSIONS USED:

ERRORS FOUND:

91 unique, 1309 total unaddressable access(es)

9 unique, 27 total uninitialized access(es)

0 unique, 0 total invalid heap argument(s)

0 unique, 0 total GDI usage error(s)

0 unique, 0 total handle leak(s)

0 unique, 0 total warning(s)

**21 unique, 69 total, 40893 byte(s) of leak(s)**

**3 unique, 3 total, 2100 byte(s) of possible leak(s)**

ERRORS IGNORED:

24 potential error(s) (suspected false positives)

(details: C:\Users\michael\AppData\Roaming\Dr. Memory\DrMemory-4-myUSBview.exe.1020.000\potential\_errors.txt)

3 potential leak(s) (suspected false positives)

(details: C:\Users\michael\AppData\Roaming\Dr. Memory\DrMemory-4-**myUSBview.exe.1020.000\potential\_errors.txt)**

**193 unique, 509 total, 28935 byte(s) of still-reachable allocation(s)**

**(re-run with "-show\_reachable" for details)**

Details: C:\Users\michael\AppData\Roaming\Dr. Memory\DrMemory-4-myUSBview.exe.1020.000\results.txt

**Итог:** конечно плачевный, **124 вида ошибок**! А в сумме **509 ошибок** при работе с памятью, **утечка около 41кбайт**!

Dr. Memory отлично справился со своей работой: по каждой ошибке дана детальная инструкция по обнаружению leak`а.

ОДНАКО, у Dr. Memory и Профилировщика MVS 2017 **результат работы разошелся**, и на мой взгляд **больше доверия вызывает Dr. Memory**, т.к. эталонная программа по его мнению освободила столько, сколько выделила, в то время как MVS 2017 не досчитался 32 байта (не много, но что-то тут не чисто).

#### 2.5.2. Анализ системных вызовов. drstrace

С помощью команды **drstrace** возможно оценить используемые системные вызовы, в формате: название вызова, его аргументы и результат исполнения;

NtQueryInformationProcess

arg 0: 0xffffffff (type=HANDLE, size=0x4)

arg 1: 0x24 (type=int, size=0x4)

arg 2: 0x0042fc9c (type=<struct>\*, size=0x4)

arg 3: 0x4 (type=unsigned int, size=0x4)

arg 4: 0x00000000 (type=unsigned int\*, size=0x4)

succeeded =>

arg 2: <NYI> (type=<struct>\*, size=0x4)

arg 4: 0x00000000 (type=unsigned int\*, size=0x4)

retval: 0x0 (type=NTSTATUS, size=0x4)

NtQueryVirtualMemory

arg 0: 0xffffffff (type=HANDLE, size=0x4)

arg 1: 0x0100687c (type=void \*, size=0x4)

arg 2: 0x0 (type=int, size=0x4)

arg 3: 0x0042fa30 (type=<struct>\*, size=0x4)

...

**Итог**: это не так привычно, как strace в Linux, но позволяет разобрать программный сбой, в случае, если другие методы не помогают. Название системных вызовов, не так очевидно, и придется серьезно покопаться… поэтому лучше использовать средства отладки и профилирования по проще.

**2.6 Профилирование с помощью gprof под Window 7 x64**

К сожалению, mingw не поддерживает современные библиотеки Win7 (а в утилите они использовались), по этой причине с основной утилитой не удастся продемонстрировать данный метод профилирования:

**g++ -g -pg uvcview.h display.h Header.h guid.h hub.h hostController.h myWinUsb.h main.cpp -o myWinUsb.exe**

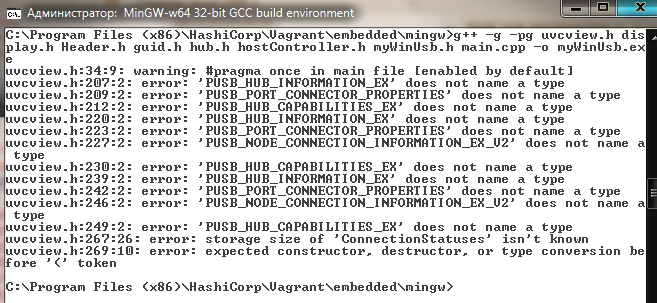
****

Рисунок 14. Компиляция в mingw myWinUSB

А так как хотелось бы рассмотреть данный метод и в Windows тоже, то вместо основной утилиты используем написанную программу Hello World (которая вообще-то не HW).

Первый этап это – компиляция в mingw:

**Компиляция в MinGw**

Setting up environment for MinGW-w64 GCC 32-bit...

Microsoft Windows [Version 6.1.7601]

(c) Корпорация Майкрософт, 2009. Все права защищены.

C:\Program Files (x86)\HashiCorp\Vagrant\embedded\mingw>

**g++ -g -pg "D:\System programming\Hello World\Hello World\main.cpp" -o hw.exe**

C:\Program Files (x86)\HashiCorp\Vagrant\embedded\mingw>

**hw.exe main.cpp**

Hit something...

Для продолжения нажмите любую клавишу . . .

Memory allocated

Hit something...

Для продолжения нажмите любую клавишу . . .

Hit something...

Memory freed

Для продолжения нажмите любую клавишу . . .

C:\Program Files (x86)\HashiCorp\Vagrant\embedded\mingw>

**gprof hw.exe gmon.out > analysis.txt**

Далее с помощью небольшой утилиты gprof2dot.py, написанной на питоне, сгенерируем модель графа из текстовых данных, командой:

**python gprof2dot.py -n0.5 -s analysis.txt > analysis.dot**

Далее, с помощью утилиты Graphviz построим граф, исполнив команду:

**dot -Tpng analysis.dot -Gcharset=latin1 -o analysis.png**

**Итог**: будет сгенерирован файл **analysis.png,** отражающий модель работы программы, с указанием наиболее ресурсоемких вызовов (рис. 15):



Рисунок 15. Граф вызовов HelloWorld

**3. LINUX**

### 3.1. Утилита strace

**Strace** - это утилита, которая **отслеживает системные вызовы**, которые представляют собой механизм трансляции, обеспечивающий интерфейс между процессом и операционной системой (ядром). Эти вызовы могут быть перехвачены и прочитаны. Это позволяет лучше понять, что процесс пытается сделать в заданное время.

Перехватывая эти вызовы, мы можем добиться лучшего понимания поведения процессов, особенно если что-то идет не так. **Функциональность операционной системы**, **позволяющая отслеживать системные вызовы, называется ptrace**. Strace вызывает ptrace и читает данные о поведении процесса, возвращая отчет.

**флаг –c** позволяет вывести информацию о количестве используемых методов:

|  |
| --- |
| % time seconds usecs/call calls errors syscall  ------ ----------- ----------- --------- --------- ----------------  38.55 0.000032 0 756 lstat  27.71 0.000023 0 430 read  18.07 0.000015 0 227 close  15.66 0.000013 0 102 12 access  0.00 0.000000 0 120 write  0.00 0.000000 0 215 open  ...  0.00 0.000000 0 1 arch\_prctl  0.00 0.000000 0 1 gettid  0.00 0.000000 0 4 1 futex  0.00 0.000000 0 1 set\_tid\_address  0.00 0.000000 0 132 44 readlinkat  0.00 0.000000 0 1 set\_robust\_list  0.00 0.000000 0 2 timerfd\_create  0.00 0.000000 0 166 getrandom  ------ ----------- ----------- --------- --------- ----------------  100.00 0.000083 2601 59 total |

**флаг -Tt** выводит время исполнения, с точностью до секунды

|  |
| --- |
| ...  23:36:14 write(1, "bMaxPacketSize0: 40\n", 20) = 20 <0.000008>  23:36:14 write(1, "\n", 1) = 1 <0.000007>  23:36:14 read(6, "\1", 1) = 1 <0.000008>  23:36:14 poll([{fd=6, events=POLLIN}, {fd=8, events=POLLIN}], 2, 0) = 0 (Timeout) <0.000006>  23:36:14 write(7, "\1", 1) = 1 <0.000006>  23:36:14 close(6) = 0 <0.000006>  23:36:14 close(7) = 0 <0.000010>  23:36:14 close(8) = 0 <0.000005>  23:36:14 write(5, "\1", 1) = 1 <0.000008>  23:36:14 futex(0x7fb103c3b9d0, FUTEX\_WAIT, 3459, NULL) = -1 EAGAIN (Resource temporarily unavailable) <0.000009>  23:36:14 close(3) = 0 <0.000013>  23:36:14 close(4) = 0 <0.000005>  23:36:14 close(5) = 0 <0.000006>  23:36:14 exit\_group(0) = ?  23:36:14 +++ exited with 0 +++ |

**флаг -e trace=NAME** – позволяет вывести информацию только об 1 системном вызове

(с именем NAME)

|  |
| --- |
| bind(3, {sa\_family=AF\_NETLINK, pid=0, groups=00000002}, 12) = 0  bind(3, {sa\_family=AF\_NETLINK, pid=0, groups=00000002}, 12) = 0  +++ exited with 0 +++ |

**Примечание**: подробное описание системного вызова в используемой системе можно получить из мануалов командой: **man 2 NAME.** Например: man 2 bind.

**3.2. Утилита ltrace**

ltrace - **регистрирует вызовы динамических библиотек** с целью отладки.

Программа ltrace предназначена **для отладки динамически собранных программ**. Отлаживаемый код запускается под управлением ltrace, при этом вызовы динамических библиотек, а также получаемые процессом сигналы перехватываются и регистрируются. Возможна также регистрация системных вызовов со стороны отлаживаемой программы.

Для отладки программы её **не нужно перекомпилировать**, поэтому возможно использование ltrace с программами, исходный текст которых не доступен.

**флаг -c** – выводит системные вызовы динамически подключаемых библиотек

|  |
| --- |
| % time seconds usecs/call calls function  ------ ----------- ----------- --------- --------------------  54.03 0.073521 36760 2 **libusb\_init**  11.45 0.015580 136 114 \_ZSt4endlIcSt11char\_...  11.45 0.015580 136 114 \_ZNSolsEPFRSoS\_E  11.00 0.014973 122 122 \_ZStlsISt11char\_...  6.84 0.009309 114 81 \_ZNSolsEi  1.41 0.001914 119 16 \_ZNSolsEt  1.39 0.001894 118 16 \_ZNSolsEPFRSt8ios\_baseS0\_E  0.70 0.000959 119 8 libusb\_get\_device\_descriptor  0.67 0.000905 452 2 libusb\_exit  0.37 0.000497 248 2 libusb\_get\_device\_list  0.34 0.000468 468 1 \_ZNSt8ios\_base4InitC1Ev  0.20 0.000270 135 2 libusb\_free\_device\_list  0.12 0.000161 161 1 \_\_cxa\_atexit  0.04 0.000055 55 1 \_ZNSt8ios\_base4InitD1Ev  ------ ----------- ----------- --------- --------------------  100.00 0.136086 482 total |

**флаг -Tttt** – отображает точное время (с точностью до микросекунды) о времени совершения вызова

|  |
| --- |
| **1488833886**.**248021** \_\_libc\_start\_main(0x400c26, 1, 0x7ffe1025a078, 0x4012d0 <unfinished ...>  **1488833886**.**248475** \_ZNSt8ios\_base4InitC1Ev(0x6021d1, 0xffff, 0x7ffe1025a088, 160) = 0 <0.000606>  **1488833886**.**249126** \_\_cxa\_atexit(0x400b10, 0x6021d1, 0x6020a0, 0x7ffe10259e50) = 0 <0.000207>  1488833886.249426 libusb\_init(0x7ffe10259f58, 0x7ffe1025a078, 0x7ffe1025a088, 192) = 0 <0.045738>  …  1488833886.406795 \_ZNSt8ios\_base4InitD1Ev(0x6021d1, 0, 0x400b10, 0x7fee24915d10) = 0x7fee24c98880 <0.000085>  1488833886.407064 +++ exited (status 0) +++ |

### 3.3. Использование gprof (есть и в Windows)

Утилита **gprof** доступна как в Linux, так и во FreeBSD. Если программа уже была собрана, делаем **make clean**. Затем **пересобираем ее с флагом -pg:**

gcc -pg -Wall htable.c htable\_test.c -o htable\_test

Запускаем программу без **gprof**:

cat test.txt | ./htable\_test > /dev/null

**Будет создан** двоичный файл **gmon.out**. Теперь запускаем программу под **gprof**:

cat test.txt | gprof ./htable\_test > profile

Полученный текстовый файл profile вполне читаемый — видно, где и сколько времени проводила программа. Так, к примеру, могут выглядеть первые несколько его строк:

|  |
| --- |
| Flat profile:  Each sample counts as 0.01 seconds.  no time accumulated  % cumulative self self total  time seconds seconds calls Ts/call Ts/call name  0.00 0.00 0.00 32 0.00 0.00 std::operator&(std::\_Ios\_Fmtflags, ...)  0.00 0.00 0.00 16 0.00 0.00 std::ios\_base::setf(std::\_Ios\_Fmtflags, ...)  0.00 0.00 0.00 16 0.00 0.00 std::operator&=(std::\_Ios\_Fmtflags&, ... )  0.00 0.00 0.00 16 0.00 0.00 std::operator~(std::\_Ios\_Fmtflags)  0.00 0.00 0.00 16 0.00 0.00 std::operator|=(std::\_Ios\_Fmtflags&, ... )  0.00 0.00 0.00 16 0.00 0.00 std::operator|(std::\_Ios\_Fmtflags, ... )  0.00 0.00 0.00 8 0.00 0.00 printInfoDevice(libusb\_device\*)  0.00 0.00 0.00 1 0.00 0.00 \_GLOBAL\_\_sub\_I\_main  0.00 0.00 0.00 1 0.00 0.00 printInfoAllDevices()  0.00 0.00 0.00 1 0.00 0.00 getCountOfAllDevices()  0.00 0.00 0.00 1 0.00 0.00 \_\_static\_initialization\_and\_destruction\_0(int, int)  ... |

Поля означают следующее:

**% time** – процентное отношение общего времени использования функции ко всему времени выполнения программы;

**call** – сколько раз данная функция была вызвана

Чтобы построить граф вызовов, который будет несомненно наглядней, следует сначала установить:

**sudo apt-get install graphviz**

**sudo pip install gprof2dot**

А затем выполнить построение графа командой:

**gprof2dot ./profile | dot -Tsvg -o output.svg**

Пример полученной картинки ( SVG, ~ 9 Кб):

### 

Рисунок 16. Граф выполнения программы

### 3.4. Профилирование при помощи perf top

**3.4.1. Основные возможности**

Установка perf в Ubuntu / Debian:

**sudo apt-get install linux-tools-common linux-tools-generic linux-cloud-tools-generic**

|  |  |
| --- | --- |
| **Вызов** | **Можно посмотреть top:** |
| sudo perf top -a | по всей системе: |
| sudo perf top -u postgres | по процессам конкретного пользователя: |
| sudo perf top -p 12345 | по конкретному процессу: |

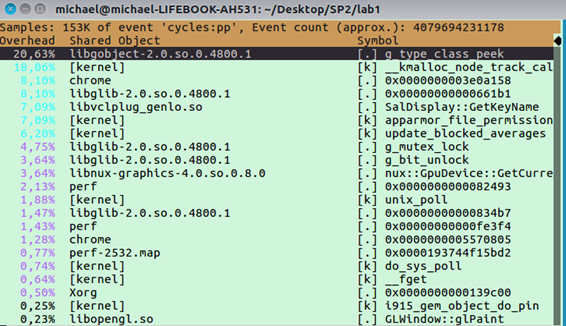


Рисунок 17. Список всех процессов

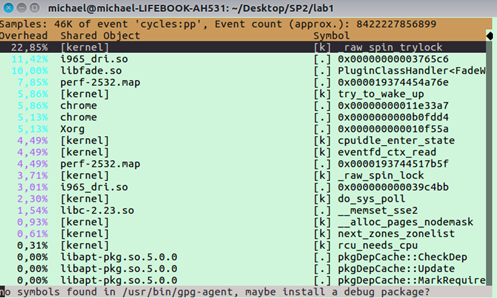


Рисунок 18. Процесс браузера «сhrome»

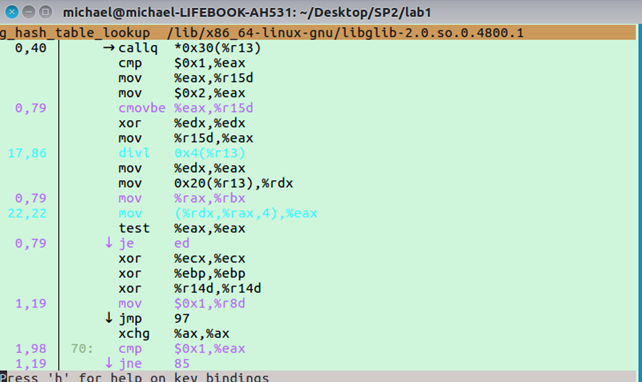


Рисунок 19. Процесс «Chrome\libglib-2.0.so.0.4800.1»

**Итоги:**

Картинка обновляется в **реальном времени**. При помощи стрелочек и клавиши Enter можно «проваливаться внутрь» процессов и функций, вплоть до подсвечивания строчек кода и ассемблерных инструкций, которые тормозят.

Во FreeBSD, насколько я понимаю, аналогичный функционал предоставляется [**pmcstat**](http://eax.me/freebsd-pmcstat/).

### 3.4.2. Строим флемграфы

Помимо отображения топа самых часто вызываемых процедур программа perf умеет много чего еще. Например, пишем stack samples с частотой 99 Герц для определенного pid’а со сборкой данных о call chains (флаг -g):

sudo perf record -p 12345 -F 99 -g

Запуск конкретной программы под perf производится так:

sudo perf record -F 99 -g -- ./myprog arg1 arg2 arg3

На выходе получаем файл perf.data. Смотрим отчет:

sudo perf report --stdio

Но читать его в таком виде не очень-то удобно. Намного удобнее построить флеймграф:

git clone https://github.com/brendangregg/FlameGraph  
sudo perf script | ./FlameGraph/stackcollapse-perf.pl > out.perf-folded  
./FlameGraph/flamegraph.pl out.perf-folded > perf.svg



Рисунок 20. Флейм граф

**Выводы:**

Dr. Memory отлично справился со своей работой: по каждой ошибке дана детальная инструкция по обнаружению leak`а.

У Dr. Memory и Профилировщика MVS 2017 результат **работы разошелся**, и на мой взгляд **больше доверия** вызывает **Dr. Memory**, т.к. эталонная программа по его мнению освободила столько, сколько выделила, в то время как MVS 2017 не досчитался 32 байт (не много, но что-то тут не чисто).

**Drstrace** - это не так привычно, как strace в Linux, но позволяет разобрать программный сбой, в случае, если другие методы не помогают. Название системных вызовов, не так очевидно, и придется серьезно покопаться… поэтому лучше использовать средства отладки и профилирования по проще.

**Perf top** позволяет наблюдать процессы в реальном времени с точки зрения обывателя. При помощи стрелочек и клавиши Enter можно «спускаться» вплоть до кода и ассемблерных инструкций. Во FreeBSD, насколько я понимаю, аналогичный функционал предоставляется **pmcstat**.

**ИСТОЧНИКИ:**

1. Метод выборки. https://www.techdays.ru/videos/2708.html

2. Как обновить/установить символьные файлы <https://msdn.microsoft.com/ru-ru/library/89axdy6y(v=vs.120).aspx>

3. Профилирование gprof (64-bit Window 7) <http://yzhong.co/profiling-with-gprof-under-64-bit-window-7/>

4.Профилирование кода на C/C++ в Linux и FreeBSD [22.03.16] <http://eax.me/c-cpp-profiling/>

5. Построение графов функций в Windows и Linux <https://github.com/jrfonseca/gprof2dot>